CoRE - Komponentenorientierte Entwicklung offener verteilter Softwaresysteme im Telekommunikationskontext

Band I - Entwicklungsprozesse und Entwicklungstechniken
Band II - Konzeptraum und Notationen
Band III - Plattformunterstützung und Ableitungsregeln für Softwarekomponenten

zur Erlangung des akademischen Grades
Doktor rerum naturalium

im Fach
Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät II
der Humboldt-Universität zu Berlin

von

Herrn Dipl.-Inf. Marc Born
geboren am 27.04.1971 in Berlin

und

Herrn Dipl.-Inf. Olaf Kath
geboren am 16.11.1969 in Prenzlau

Präsident der Humboldt-Universität zu Berlin

Prof. Dr. Jürgen Mlynek

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II

Prof. Dr. Elmar Kulke

Gutachter

1. Prof. Dr. Joachim Fischer, Humboldt-Universität zu Berlin
2. Prof. Dr. Dr. h.c. Radu Popescu-Zelletin, Technische Universität Berlin/Fraunhofer Gesellschaft
3. Prof. Dr. Lambert J. M. Nieuwenhuis, University of Twente

Tag der mündlichen Prüfung: 29.04.2002

CoRE - Komponentenorientierte Entwicklung offener verteilter Softwaresysteme im Telekommunikationskontext

Band I - Entwicklungsprozesse und Entwicklungstechniken

Marc Born

Fraunhofer-Institut für Offene Kommunikationssysteme

Kaiserin-Augusta-Allee 31
10589 Berlin

born@fokus.fhg.de
Tel. +49 (30) 3463 7235
Fax. +49 (30) 3463 8235

Olaf Kath

Humboldt-Universität zu Berlin
Institut für Informatik

Rudower Chaussee 25
12489 Berlin

kath@informatik.hu-berlin.de
Tel. +49 (30) 2093 3117
Fax. +49 (30) 2093 3112

ZUSAMMENFASSUNG

Die Telekommunikation und die ihr zuliefernde Industrie stellen einen softwareintensiven Bereich dar - der durch einen sehr hohen Anteil von Eigenentwicklungen gekennzeichnet ist. Eine wesentliche Ursache dafür sind spezielle Anforderungen an Telekommunikationssoftwaresysteme, die i.allg. nicht durch Standardsoftwareprodukte sichergestellt werden können. Diese Anforderungen ergeben sich aus den besonderen Eigenschaften solcher Softwaresysteme wie die Verteilung der Komponenten von Softwaresystemen sowie die Verteilung der Entwicklung dieser Komponenten, die Heterogenität der Entwicklungs- und Einsatzumgebungen für diese Komponenten und die Komplexität der entwickelten Softwaresysteme hinsichtlich nichtfunktionaler Charakteristika. Die *industrielle* Entwicklung von Telekommunikationssoftwaresystemen ist ein schwieriger und bisher nicht zufriedenstellend gelöster Prozeß. Aktuelle Forschungsarbeiten thematisieren Softwareentwicklungsprozesse und -techniken sowie unterstützende Werkzeuge zur Erstellung und Integration wiederverwendbarer Softwarekomponenten („*Componentware*“).

Das Ziel dieser Dissertation besteht in der Unterstützung der industriellen Entwicklung offener, verteilter Telekommunikationssoftwaresysteme. Dazu wird die Entwicklungstechnik Objektorientierte Modellierung mit dem Einsatz von Komponentenarchitekturen durch die automatische Ableitung von Softwarekomponenten aus Modellen zusammengeführt. Die zentrale Idee ist dabei eine präzise Definition der zur Entwicklung von verteilten Softwaresystemen einsetzbaren Modellierungskonzepte in Form eines Metamodells. Von diesem Metamodell ausgehend werden dann zur Konstruktion und Darstellung objektorientierter Entwurfsmodelle eine graphische und eine textuelle Notation definiert. Da die Notationen die Konzepte des Meta-Modells visualisieren, haben sie diesem gegenüber einen sekundären Charakter. Für die Transformation von Entwurfsmodellen in ausführbare Applikationen wurde auf der Grundlage von CORBA eine Komponentenplattform realisiert, die zusätzlich zu Interaktionen zwischen verteilten Softwarekomponenten auch Entwicklungs-, *Deployment*- und Ausführungsaspekte unterstützt. Wiederum ausgehend vom Metamodell wird durch die Anwendung wohldefinierter Ableitungsregeln die automatische Überführung von Entwurfsmodellen in Softwarekomponenten des zu entwickelnden Systems ermöglicht. Die von den Autoren erarbeiteten Konzeptionen und Vorgehensweisen wurden praktisch in eine Werkzeugumgebung umgesetzt, die sich bereits erfolgreich in verschiedenen Softwareentwicklungsprojekten bewährt hat.

SUMMARY

The telecommunication industry and their suppliers form a software intensive domain. In addition, a high percentage of the software is developed by the telecommunication enterprises themselves. A main contributing factor for this situation are specific requirements to telecommunication software systems which cannot be fulfilled by standard off-the-shelf products. These requirements result from particular properties of those software systems, e.g. distributed development and execution of their components, heterogeneity of execution and development environments and complex non-functional characteristics like scalability, reliability, security and manageability. The development of telecommunication software systems is a complex process and currently not satisfactory realized. Actual research topics in this arena are software development processes and development techniques as well as tools which support the creation and integration of reusable software components (component ware).

The goal of this thesis work is the support of the industrial development and manufacturing of open distributed telecommunication software systems. For that purpose, the development technique object oriented modelling and the implementation technique usage of component architectures are combined. The available modelling concepts are precisely defined as a metamodel. Based on that metamodel, graphical and textual notations for the presentation of models are developed. To enable a smooth transition from object oriented models into executable components a component architecture based on CORBA was also developed as part of the thesis. This component architecture covers besides the interaction support for distributed components deployment and execution aspects. Again on the basis of the metamodel code generation rules are defined which allow to automate the transition from models to components

The development techniques described in this thesis have been implemented as a tool chain. This tool chain has been successfully used in several software development projects.

VORWORT

Der Markt für Softwaresysteme und die damit verbundene Entwicklung der Komponenten dieser Systeme ist bereits heute ein wirtschaftlicher Schlüsselbereich der führenden Industrienationen. Im Jahre 1999 erreichte dieser Markt allein in Deutschland ein Volumen von 55,5 Mrd. DM bei jährlichen Wachstumsraten von 12% [Bitk 00]. Der Einsatz von Softwaresystemen und deren Entwicklung bestimmen die Konkurrenzfähigkeit der Unternehmen - Experten und Branchenvertreter erwarten, daß sich dieser Markttrend auch in den kommenden Jahren fortsetzen wird [Bitk 00a]. Insbesondere die Telekommunikation stellt dabei einen äußerst softwareintensiven Bereich dar - der zudem durch einen im Vergleich zu anderen Branchen sehr hohen Anteil der Eigenentwicklungen von Softwaresystemen durch Telekommunikationsunternehmen gekennzeichnet ist (59% nach [BMBF00]). Eine wesentliche Ursache dafür ist die durch diese Firmen getroffene Einschätzung, daß Software in der Telekommunikation in hohem Maße zum Alleinstellungsmerkmal der Unternehmen beiträgt – zukünftige Leistungsmerkmale wollen sie selbst entwickeln. Mehr noch, spezifische Eigenschaften von Telekommunikationssoftwaresystemen, lassen sich oftmals nicht durch Standardsoftwareprodukte sicherstellen. Zu diesen Eigenschaften gehören [Sun 01][EU P910]:

- Verteilung der Komponenten von Softwaresystemen sowie Verteilung der Entwicklung dieser Komponenten,
- Heterogenität der Entwicklungs- und Einsatzumgebungen für diese Komponenten,
- Komplexität der entwickelten Softwaresysteme hinsichtlich nicht-funktionaler Charakteristika, wie z.B. Skalierbarkeit, Verfügbarkeit, Sicherheit und Management.

Die Entwicklung von Softwaresystemen mit diesen Eigenschaften ist immer noch ein schwieriger und nicht zufriedenstellend gelöster Prozeß, der i.allg. eine große Anzahl von verschiedenen Arbeitsschritten umfaßt. Die reine Implementierung der Softwarekomponenten nimmt dabei einen vergleichsweise kleinen Anteil an der Entwicklungszeit ein. Eine durch das Bundesministerium für Bildung und Forschung beauftragte Studie [BMBF 00] identifiziert im Kontext der Entwicklung von Softwaresystemen Forschungsschwerpunkte, die

insbesondere für die Entwicklung von Telekommunikationssoftwaresystemen relevant sind. Zu diesen gehören Softwareentwicklungsprozesse und -techniken sowie begleitende Werkzeuge zur:

- Erstellung und Integration wiederverwendbarer Softwarekomponenten („*Componentware*“) auf der Basis langlebiger Softwarearchitekturen,
- rationellen Erstellung von Software hoher Qualität,
- Erstellung räumlich verteilter, konfigurierbarer und skalierbarer Softwaresysteme in heterogenen Anwendungsumgebungen.

Ein grundsätzliches Ziel der Forschung im Bereich von Softwareentwicklungsprozessen ist die Umsetzung einer langjährigen Vision im Hinblick auf eine *industrielle* Fertigung von Softwaresystemen. Dabei fixieren diese Prozesse die auszuführenden Arbeitsschritte, deren Resultate und Zusammenhänge (Abläufe). Industrialisierung von Softwareentwicklungsprozessen bedeutet Automatisierung der Arbeitsschritte sowie der Übergänge zwischen diesen Schritten. Dabei werden die Resultate eines Arbeitsschrittes für die nachfolgenden Arbeitsschritte *automatisiert* nutzbar gemacht.

Während der einzelnen Arbeitsschritte und bei deren Übergängen wird die notwendige Automatisierung durch die Anwendung von Entwicklungstechniken und diese unterstützende Werkzeuge erreicht. Beispiele für in diesem Zusammenhang allgemein verwendete Entwicklungstechniken sind die Erstellung von Pflichten- und Lastenheften, die Modellierung und Simulation, die Codegenerierung und das funktionale Testen von Softwarekomponenten. Aufgrund der Charakteristika von Softwaresystemen im Telekommunikationsbereich sind sowohl die anzuwendenden Entwicklungsprozesse als auch die unterstützenden Entwicklungstechniken, mit denen Softwaresysteme in diesem Bereich entwickelt werden, zu präzisieren.

Es gibt bereits eine Fülle von Softwareentwicklungsprozessen, die im industriellen Umfeld zum Einsatz kommen - durch deren Weiterentwicklung und Spezialisierung allein lassen sich die genannten Forschungsschwerpunkte nicht zufriedenstellend bearbeiten [Bitk 00a]. Statt dessen muß besonderer Wert auf die *Weiterentwicklung und Spezialisierung der Entwicklungstechniken* gelegt werden, die in *unterschiedlichen* Softwareentwicklungsprozessen zum Einsatz kommen können.

Eine Analyse im Bereich der Telekommunikation [Gart99] ergab, daß in zunehmendem Maße die Entwicklungstechniken *objektorientierte Modellierung* sowie *Einsatz von Komponentenarchitekturen* zur Integration von Systemkomponenten zur Anwendung kommen. Komponentenarchitekturen stellen die Infrastruktur für die Kommunikation zwischen den Komponenten eines Softwaresystems bereit. Beispiele für Komponentenarchitekturen im Bereich Telekommunikation sind *Common Object Request Broker Architecture* (CORBA, [OMG CORBA]) und *Enterprise Java Beans* (EJB, [SunEJB]). Die Integration dieser Entwicklungstechniken ist aber bisher nur rudimentär erfolgt. Als Ursache dafür wird von [Gart99] die bislang nur partiell mögliche automatische Überführung von Modellen in Softwarekomponenten gesehen. Die Möglichkeiten der objektorientierten Modellierung lassen sich somit nicht umfassend nutzen, die Erstellung von umfangreichen Modellen ist bezogen auf die Entwicklungszeit ineffizient.

Zur Unterstützung der objektorientierten Modellierung für Telekommunikationssoftwaresysteme bedarf es einer Präzisierung der Konzepte und Beziehungen, die zur Modellbildung herangezogen werden können. Die Basiskonzepte sind allgemein bekannt und repräsentieren Paradigmen der Objektorientierung (z.B. Klassifizierung, Komposition und Generalisierung). Diese allgemeinen Konzepte können natürlich auch im Bereich der Modellierung von Telekommunikationssoftwaresystemen angewendet werden. Darüber hinaus sind jedoch domänenspezifische Konzepte und Beziehungen einzubeziehen, die die speziellen Anforderungen des Telekommunikationsbereiches zu berücksichtigen gestatten (z.B. verteilte Objekte, multimediale Interaktionen, Konfigurationskonzepte, Gütebeschreibung und -garantie).
In dieser Arbeit werden - basierend auf internationalen Standards, wissenschaftlichen und industriellen Ansätzen sowie Erfahrungen der Autoren aus einer Reihe von Industrieprojekten und internationalen Kooperationen - Modellierungskonzepte und -beziehungen konstruiert. Dabei wird die Erweiterbarkeit der entstehenden Menge von Modellierungskonzepten und -beziehungen (hier als Konzeptraum bezeichnet),

eine Anforderung, die sich aus der auch weiterhin zu erwartenden Dynamik des Bereiches Telekommunikation ergibt, durch die Anwendung geeigneter Technologien sichergestellt.

Zur Darstellung objektorientierter Modelle, die auf diesen Konzepten und Beziehungen basieren, müssen geeignete Darstellungsformen (Notationen) bereitgestellt werden. Entwickler, die objektorientierte Modellierung einsetzen, haben i.allg. subjektive Anforderungen an und Präferenzen für bestimmte Notationsformen (z.B. graphische vs. textuelle Notation, präskriptive vs. deskriptive Notation). Es ist also zweckmäßig, unterschiedliche Notationen bereitzustellen. Dann müssen aber Modelle, die mittels verschiedener Notationen dargestellt sind, ineinander überführbar sein. Es ist also offensichtlich zweckmäßig, Notationen grundsätzlich basierend auf vorher erklärten Konzepten und Beziehungen zu definieren - und nicht umgekehrt.
In dieser Arbeit werden exemplarisch zwei Notationen vorgestellt. Es wird gezeigt, wie verschieden dargestellte Modelle ineinander überführt werden können.

Der Einsatz von Komponentenarchitekturen findet in zunehmenden Maße in Unternehmen des Telekommunikationsbereiches Verwendung [Gart99a]. Im Telekommunikationsbereich verwendete Komponentenarchitekturen müssen - über die Bereitstellung eines reinen Kommunikationsmechanismus hinaus - bezüglich der nicht-funktionalen Anforderungen von Softwaresystemen und der Unterstützung der Konfigurierbarkeit und des Managements dieser Systeme erweitert werden.
In dieser Arbeit werden existierende Komponentenarchitekturen hinsichtlich der für Telekommunikationssoftwaresysteme relevanten Eigenschaften analysiert. Auf der Grundlage der weit verbreiteten Komponentenarchitektur CORBA wird exemplarisch gezeigt, wie notwendige Erweiterungen durch Einführung einer technologieunabhängigen Zwischenschicht realisiert werden können.

Zur vollständigen Integration der Entwicklungstechniken objektorientierte Modellierung und Einsatz von Komponentenarchitekturen ist die automatische Überführung von Modellen in Softwarekomponenten des zu entwickelnden Systems auf der Basis der verwendeten Komponentenarchitektur essentiell. Gerade aus der automatischen Überführbarkeit ergibt sich die Flexibilität dieser Integration [OMG MDA]:
„*[...] the ability to derive code from a stable model as the underlying infrastructure*[1] *shifts over time. ROI*[2] *flows from the reuse of application and domain models across the software lifespan.*“.
In dieser Arbeit werden Ableitungsregeln definiert, die Modelle auf der Basis der Menge der Konzepte und Beziehungen automatisch in Softwarekomponenten zu überführen gestatten. Dabei wird die realisierte technologieunabhängige Zwischenschicht als Zielumgebung der Ableitung adressiert. Die Zielumgebung unterstützt über die Kommunikationsinfrastruktur für Softwarekomponenten hinaus nicht-funktionale Anforderungen, Konfigurierbarkeit und Management von Telekommunikationssoftwaresystemen.

Die vorliegende Arbeit faßt die Ergebnisse einer zweijährigen intensiven Forschungskooperation der Autoren zusammen. Dabei brachten sie Erfahrungen aus unterschiedlichen Arbeitsbereichen und Forschungsschwerpunkten ein. Marc Born beschäftigte sich vordergründig mit der Entwicklungstechnik objektorientierte Modellierung verteilter Telekommunikationssoftwaresysteme sowie der Entwicklung von geeigneten Notationen. Olaf Kath befaßte sich stärker mit Komponentenarchitekturen und der automatischen Ableitung von Softwarekomponenten. Beide Autoren haben die Synergieeffekte einer Integration beider Schwerpunkte erkannt und durch gemeinschaftliche Tätigkeit ausgenutzt: So wurde der telekommunikationsspezifische Konzeptraum zunächst gemeinsam entworfen. Die von Herrn Born vorgenommene Präzisierung und Formalisierung dieses Konzeptraumes und dessen technologische Realisierung dienten Herrn Kath als Ausgangspunkt für die Definition und Implementierung der Regeln zur automatischen Ableitung von Softwarekomponenten, deren Zielumgebung ebenfalls von Herrn Kath als Komponentenarchitektur federführend entwickelt worden ist. Ergänzt um die von Herrn Born erarbeiteten Notationen konnte somit die vollständige Integration der objektorientierten Modellierung mit dem Einsatz von Komponentenarchitekturen geschaffen werden. Diese modellzentrierte Arbeitsweise erfordert gerade die gemeinschaftliche Erarbeitung der konzeptionellen Basis für die Entwicklungstechniken. Der Nachweis

1. Der Begriff Infrastruktur wird in MDA als Synonym für Komponentenarchitektur genutzt.
2. *Return of Investment*

der Praktikabilität dieses Ansatzes und der Eignung des Konzeptraumes in der Telekommunikationsdomäne wurde durch die prototypische Realisierung der präsentierten Entwicklungstechniken [BK 01a] und deren Anwendung in Softwareentwicklungsprojekten [EU P924][ComPoTel00] erbracht.

Die Darstellung der Ergebnisse erfolgt in drei Bänden:

- In gemeinsamer Arbeit werden die Grundlagen der erarbeiteten Entwicklungstechniken, die an diese gestellten Anforderungen im Bereich der Telekommunikation, eine Analyse ihrer Nutzbarkeit im Kontext verschiedener Softwareentwicklungsprozesse sowie die informale Beschreibung der Grundkonzepte und Beziehungen der Entwicklungstechnik objektorientierte Modellierung im Band I von den Autoren dargestellt.
- Die Verfeinerung und formalisierte Konstruktion des Konzeptraumes, die Auswahl einer dazu geeigneten Technologie, die Analyse und Bereitstellung von verschiedenen Notationen sowie deren Integration wird im Band II [CoRE II] von Marc Born ausgeführt.
- Die Analyse einsetzbarer Komponentenarchitekturen, die Realisierung der technologieunabhängigen Zwischenschicht, die Definition der Ableitungsregeln für Softwarekomponenten sowie die Darstellung relevanter Interaktionsabläufe sind Gegenstand der Untersuchungen von Olaf Kath in Band III [CoRE III].

Der erste Band schließt mit einem von beiden Autoren erarbeiteten Resümee, das eine Diskussion grundsätzlicher Aspekte der Ergebnisse der Gesamtarbeit beinhaltet und diese Ergebnisse in einen größeren wissenschaftlichen und technologischen Zusammenhang einordnet. In den nachfolgenden Bänden wird diese Diskussion um spezifische Aspekte erweitert. In den Vorworten der Bände II und III werden die Abhängigkeiten zu den in den jeweils anderen Bänden dargestellten Sachverhalten aufgezeigt und die Ziele des jeweiligen Bandes präzisiert. Die Lektüre des ersten Bandes, insbesondere der Darstellung des Konzeptraumes, ist für das Verständnis der gesamten Arbeit essentiell. Die nachfolgenden Bände sind dann unabhängig voneinander lesbar: Leser, deren Interesse das Verständnis der Mechanismen der Komponentenarchitektur und der Regeln zur Ableitung von Softwarekomponenten für diese Architektur gilt, seien auf Band III verwiesen. Besteht das Interesse des Lesers vorwiegend im tiefen Verständnis des Konzeptraumes, insbesondere der Zusammenhänge zwischen den Konzepten, so ist die Lektüre von Band II notwendig. Für einen Einsatz der realisierten Entwicklungswerkzeuge (s. [BK 01a]) ist die Lektüre von Band II, Kapitel 5 (Notationen) sowie das Verständnis von Band III, Kapitel2 erforderlich. Allen Bänden sind ein Abkürzungsverzeichnis, ein Index und ein umfassendes Literaturverzeichnis beigefügt.

Englische Begriffe, für die keine adäquate Übersetzung gefunden wurde, sind *kursiv* geschrieben. Detaillierte Kenntnisse des Lesers über UML ([OMG UML1.3]), CORBA-IDL ([OMG CORBA IDL]) sowie C++ ([ANSI C++]) werden für alle Bände vorausgesetzt.

Unser besonderer Dank gilt Herrn Prof. Dr. J. Fischer, der diese Arbeit betreut und ihre Durchführung ermöglicht hat. Seine zahlreichen Ratschläge und die organisatorische Unterstützung haben maßgeblich zu ihrem Erfolg beigetragen. Herrn Prof. Dr. Dr. h. c. R. Popescu-Zeletin sei ebenfalls herzlich für seine Betreuung und die gelegentlich notwendige Aufmunterung gedankt. Das Gewähren von Freiräumen durch ihn war essentiell für die erfolgreiche Fertigstellung unter den gegebenen zeitlichen Randbedingungen. Herrn Prof. Dr. L. J. M. Nieuwenhuis sei für die Übernahme des Korreferates herzlichst gedankt.

Weiterhin danken wir allen Kolleginnen und Kollegen des Instituts für Informatik der Humboldt-Universität zu Berlin und GMD Fokus für ihre intensive und geduldige Unterstützung. Namentlich erwähnen möchten wir Herrn H. Böhme, Herrn Dr. E. Holz, Herrn M. v. Löwis, Herrn Dr. E. Möller, Herrn B. Neubauer, Herrn T. Ritter, Herrn F. Stoinski, Frau Dr. I. Schieferdecker, Herrn G. Schürmann, Frau L. Strick und Herrn Dr. M. Tschichholz. Ganz besonderer Dank gilt Frau M. Albrecht, Herrn Dr. K. Ahrens und Herrn M. Hagen für die organisatorische Unterstützung.

Die größte Stütze für Olaf war seine Familie - Claudia hat mit viel Geduld, Hilfe und Toleranz die endlosen Abende und vielen Reisen ertragen, die dieser Arbeit gewidmet waren. Ich bin dankbar, daß sie diese Arbeit so unterstützt hat. Meiner Tochter Hanna sei versprochen, daß die versäumte Zeit zum Spielen so bald als

möglich nachgeholt wird. Auch meinen Eltern und meinen Freunden sei an dieser Stelle für ihr Verständnis und ihre aufmunternden Worte gedankt, die über unvermeidliche Durststrecken hinweggeholfen haben.

Der besondere Dank von Marc gilt an dieser Stelle seiner Mutter, die durch ihren Einsatz und ihren Beistand diese Arbeit unterstützt hat. Nicht vergessen möchte ich auch meine Freunde, die mich oft aufgemuntert und trotz der durch die vielen Arbeitsabende und Reisen knappen gemeinsamen Stunden nicht vergessen haben.

Berlin, im Juli 2001

Marc Born, Olaf Kath
http://www.dot-profile.de
born@fokus.fhg.de
kath@informatik.hu-berlin.de

INHALT

KAPITEL 1

Einführung und Grundbegriffe

Grundanliegen dieser Arbeit ist die Unterstützung der Entwicklung von verteilten Softwaresystemen im Telekommunikationskontext.

Softwaresysteme sind dadurch gekennzeichnet, daß ihre Systembestandteile (Komponenten) gerade Software sind (Softwarekomponenten), die zielgerichtet miteinander interagieren. In verteilten Softwaresystemen sind die Softwarekomponenten räumlich verteilt. Zur Erfüllung des Systemszweck müssen diese Softwarekomponenten auf dafür geeigneten Maschinen bereitgestellt und ausgeführt werden. Bei der Bereitstellung wird ein physikalischer Repräsentant (z.B. eine Binärdatei) der Softwarekomponente auf der für die Ausführung vorgesehenen Maschine verfügbar gemacht sowie die Voraussetzung für die Ausführung (z.B. durch Konfiguration) geschaffen. Das Ziel der Entwicklung von Softwaresystemen besteht in der Fabrikation von Softwarekomponenten, deren Integration sowie Ausführung zur Verwirklichung des Systemzwecks.

Die Entwicklung von Softwaresystemen, insbesondere von verteilten Softwaresystemen, ist ein komplexer und zeitaufwendiger Prozeß. Um diesen Prozeß mit ingenieurtechnischen Mitteln zu unterstützen, etablierte sich ein allgemein anerkanntes Modell, das den Entwicklungsprozeß in aufeinanderfolgende Phasen zerlegt (Phasenmodell). Initial erfolgte diese Zerlegung in die Phasen Analyse und Definition, Entwurf, Implementierung sowie Integration und Test [Roy 70]. Später wurde dieses Modell um die Phase Einsatz und Wartung ergänzt (vgl. [PS94]). In den einzelnen Phasen sind nach diesem Modell jeweils bestimmte Aspekte des zu entwickelnden Softwaresystems zu untersuchen. Die Definition des Phasenmodells schließt die Präzisierung der Ziele der einzelnen Phasen und der in diesen zu produzierenden Ergebnisse ein. Die präzise definierten Ergebnisse einer Phase sind Voraussetzung für das Erreichen der Ziele der nachfolgenden Phasen. Der Vorteil des Phasenmodells besteht darin, daß das Gesamtproblem in weniger komplexe Teilprobleme zerlegt werden kann. Mit der Aufteilung in Phasen wird zudem die Handhabbarkeit des Problems Softwareentwicklung erhöht, da klassische Techniken des Managements wie Meilensteine, Zwischenberichte und Qualitätssicherung systematisch auf die Phasen und deren Ergebnisse angewendet werden können.

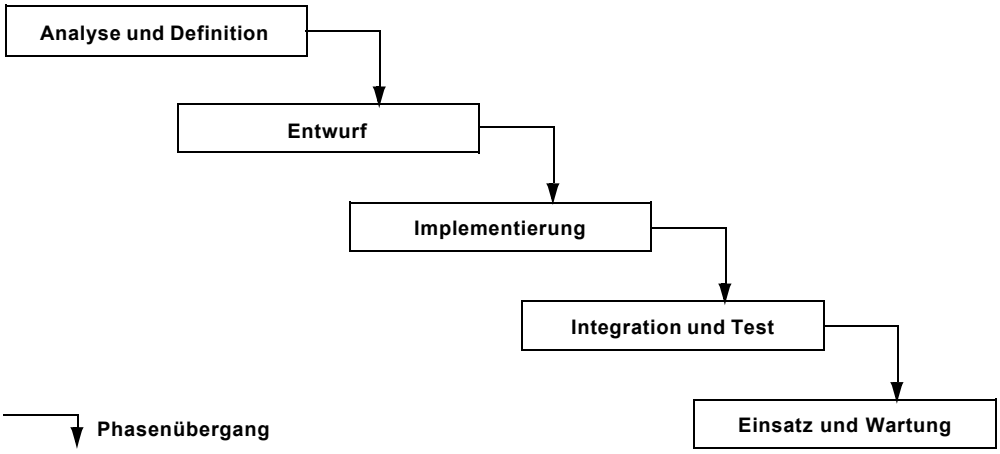


Abb. 1 Das klassische Phasenmodell der Softwareentwicklung

Da das Verständnis des klassischen Phasenmodells (vgl. Abb.1) essentiell für die Positionierung dieser Arbeit ist, werden hier die Phasen und ihre Ergebnisse im Überblick dargestellt.

- *Analyse und Definition*
  In dieser Phase werden die Anforderungen an das Softwaresystem ermittelt, eine Kosten-Nutzen-Analyse vorgenommen sowie eine Durchführbarkeitsstudie erstellt. Diese Phase beschäftigt sich ausschließlich mit der Außenwirkung des Softwaresystems, also seinem Systemzweck.
  Wichtigstes Ergebnis bezüglich der nachfolgenden Phasen ist die Definition der Anforderungen an das Softwaresystem.
- *Entwurf*
  In dieser Phase wird die Struktur des Softwaresystems durch funktionale Dekomposition festgelegt sowie nicht-funktionale Anforderungen mit dieser Struktur verbunden. Diese Dekomposition basiert auf den Anforderungen an das Softwaresystem und schließt die Definition des Zusammenwirkens, also der zielgerichteten Interaktion der resultierenden Komponenten ein.
  Ergebnis ist die Spezifikation des Softwaresystems, die unabhängig von einer konkreten programmiersprachlichen Umsetzung ist.
- *Implementierung*
  In dieser Phase erfolgt die Realisierung der Spezifikation des Softwaresystems unter Nutzung einer oder mehrerer Programmiersprachen.
  Ergebnis sind implementierte, dokumentierte und separat getestete Softwarekomponenten. Die in dieser Phase erfolgten Tests schließen allerdings nicht den Test des Zusammenwirkens der Softwarekomponenten ein.
- *Integration und Test*
  In dieser Phase werden die Softwarekomponenten integriert, d.h. das Softwaresystem wird zusammengesetzt. Diese Integration wird von Tests begleitet, die das korrekte Zusammenwirken der Softwarekomponenten bezüglich der definierten Anforderungen überprüfen.
  Wichtigstes Ergebnis dieser Phase ist der erfolgreiche Abschlußtest, also der Nachweis, daß das Softwaresystem in der Lage ist, die gestellten Anforderungen zu erfüllen.
- *Einsatz und Wartung*
  In dieser Phase wird das Softwaresystem in seiner Zielumgebung bereitgestellt und ausgeführt. Auftretende Fehlfunktionen werden behoben, Anpassungen an veränderte Umgebungen oder zusätzliche Funktionalitäten werden realisiert.

Diese Phase liefert Erfahrungen und Resultate der Verwendung des Softwaresystems unter Einsatzbedingungen sowie Ideen für zukünftig zu entwickelnde Softwaresysteme.

Wegen seiner sequentiellen Vorgehensweise wird das vorgestellte Phasenmodell auch als Wasserfallmodell bezeichnet. Die Praxis hat jedoch gezeigt, daß ein derart idealer, d.h. streng sequentieller Ablauf der Softwareentwicklung i.allg. nicht zu erreichen ist. Treten z.B. in der Implementierungsphase Probleme auf, so macht deren Lösung u.U. Modifikationen der Spezifikation erforderlich. Dies ist dann der Fall, wenn die Struktur des Systems geändert oder zusätzliche, bisher nicht berücksichtigte Beziehungen zwischen Komponenten hinzugefügt werden müssen. Die Erkenntnis dieses Schwachpunkts hat zur Verfeinerung des Phasenmodells durch die Definition von Iterationsmöglichkeiten geführt [Pom91]. Dabei wird davon ausgegangen, daß sowohl innerhalb der einzelnen Phasen als auch zwischen den Phasen Iterationen stattfinden können. Weitere Abwandlungen des klassischen Phasenmodells beziehen explizit die Erstellung von Vorabversionen des Softwaresystems (Prototypen) in den Entwicklungsprozeß mit ein. Eine Variante des Phasenmodells mit Iterationen und ohne explizite Prototypen ist in Abb. 2 dargestellt.

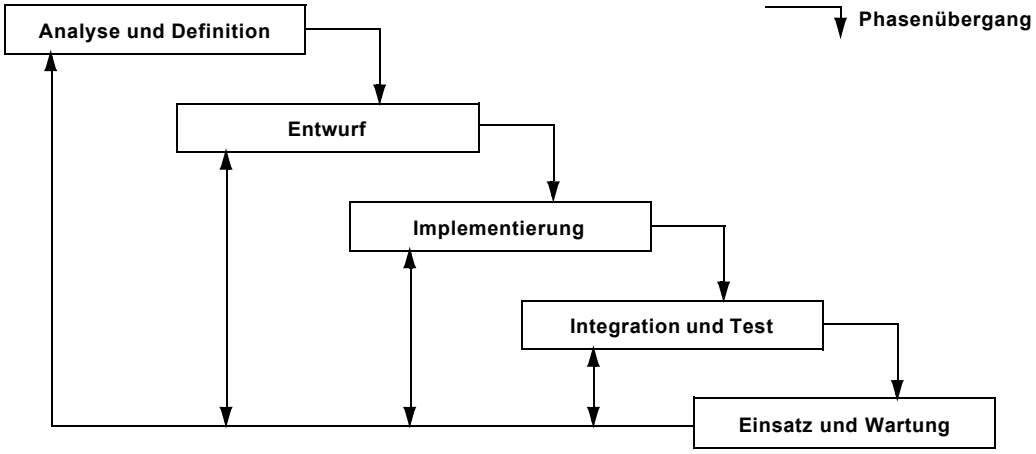


Abb. 2 Das Phasenmodell der Softwareentwicklung mit Iterationen

Generelle Zielsetzung aller Varianten des Phasenmodells ist die Entwicklung von Softwaresystemen mit hoher Qualität bei planbaren Kosten in einem ebenfalls planbaren Zeitrahmen. Um diese generelle Zielsetzung zu erreichen, werden in den einzelnen Phasen Entwicklungstechniken eingesetzt. Entwicklungstechniken sind Hilfsmittel, um die Ergebnisse der Phase zeit- und kostengünstig sowie qualitativ hochwertig bereitzustellen. Insbesondere gehören Entwicklungswerkzeuge zu diesen Hilfsmitteln, die zur Automatisierung der Entwicklungstechniken genutzt werden. Betrachtet man den iterativen Entwicklungsprozeß nach Abb.2, so wird deutlich, daß Entwicklungstechniken nicht nur innerhalb der einzelnen Phasen eingesetzt werden können, sondern insbesondere beim Übergang zwischen den einzelnen Phasen, da diese Übergänge durch Iterationen häufiger ausgeführt werden und einen nicht unwesentlichen Anteil an der Entwicklungszeit einnehmen. Ziel des Einsatzes von Entwicklungstechniken ist die Automatisierung der in den einzelnen Phasen vorzunehmenden Arbeitsschritte sowie der Phasenübergänge durch Werkzeuge. Ein derart automatisierter Übergang ist einerseits effektiv durchführbar und verspricht andererseits, daß alle Informationen der vorausgehenden Phase auch in die nachfolgenden Phasen einfließen. Es ist also zu untersuchen, welche Übergänge sich für eine Werkzeugunterstützung eignen.

- *Von Analyse und Definition zu Entwurf*
  Die Analyse und Definitionsphase liefert die Anforderungsdefinition. Ein automatisierter Übergang von der Anforderungsdefinition in eine Systemspezifikation ist derzeit nicht möglich. Die Ableitung konkreter Dekompositionsdefinitionen aus den Anforderungen ist ein wertschöpferischer, kreativer Prozeß - die Ableitung einer inneren Struktur des Softwaresystems aus der Außenwirkung desselben. Allerdings

kann die Verwaltung der Anforderungen und die Verbindung der Systemspezifikation mit den Anforderungen (also die Konsistenzsicherung) werkzeugunterstützt erfolgen.

(Beispiel 1) *Use-Case*-Diagramme [OMG UML1.3] können eingesetzt werden, um typische Anwendungsfälle für ein zu entwickelndes Softwaresystem zu erfassen. Diese Anwendungsfälle können durch *Trace*-Relationen mit der Spezifikation des Softwaresystems verbunden werden, um beispielsweise bei Änderungen der Spezifikation feststellen zu können, welche Anwendungsfälle hiervon betroffen sind.

- *Von Entwurf zu Implementierung*
  Systemspezifikationen lassen sich in Implementierungsgerüste überführen. Dazu werden die Beschreibungen der Systembestandteile und der Interaktionen zwischen diesen in programmiersprachliche Konstrukte überführt. Diese werden nachfolgend vervollständigt und zu Softwarekomponenten zusammengesetzt. Die Überführung der Systemspezifikation in programmiersprachliche Konstrukte ist werkzeugunterstützt durchführbar.

  (Beispiel 2) Die Schnittstellen der Komponenten eines verteilten Softwaresystems, das auf der Basis von CORBA (*Common Object Request Broker Architecture*, [OMG CORBA]) entwickelt wird, werden mittels CORBA-IDL (*Interface Definition Language*) definiert. Diese Schnittstellendefinitionen sind Bestandteil der Systemspezifikation, sie können mittels *Compiler*-Werkzeugen in programmiersprachliche Konstrukte überführt werden.

- *Von Implementierung zu Integration und Test*
  Softwarekomponenten als Ergebnis der Implementierungsphase können durch Anwendung von geeigneten Mechanismen so implementiert werden, daß sich ihre spätere, im Softwaresystem resultierende Integration vereinfacht. Diese Mechanismen müssen in der Implementierungsphase bereitgestellt und benutzt werden, die Benutzung muß wiederum durch den Übergang von der Spezifikation zu den programmiersprachlichen Konstrukten der Implementierung vorbereitet sein. Die Werkzeugunterstützung für den Übergang von Implementierung zu Integration und Test beginnt also bereits beim Übergang vom Entwurf zur Implementierung.

  (Beispiel 3) Eine während der Integrationsphase verwendete Komponentenarchitektur wie CORBA stellt Mechanismen zur Unterstützung der asynchronen Kommunikation zwischen Softwarekomponenten bereit (CORBA-*Notification*-Service, [OMG CORBA NoS]). Falls in der Spezifikation die asynchrone Kommunikation vorgesehen und in der Implementierungsphase realisiert wird, können diese Spezifikationsteile automatisch auf die Mechanismen der Komponentenarchitektur abgebildet und zur Implementierung genutzt werden.

- *Von Integration und Test zu Einsatz und Wartung*
  Die Einsatzumgebung eines Softwaresystems hat oft andere Charakteristika als die Entwurfs-, Implementierungs- und Integrationsumgebung. Das ist dann der Fall, wenn der Einsatz eines Softwaresystems in einem oder mehreren Unternehmen erfolgen soll, die die vorangegeangenen Entwicklungsphasen nicht selbst durchgeführt haben. Der Übergang eines Softwaresystems von der Integrations- in die Einsatzumgebung läßt sich werkzeugunterstützt ausführen. Dazu müssen zusätzlich zu den Softwarekomponenten Beschreibungen über deren Anforderungen an die Einsatzumgebung und ihre Integration erstellt werden, die dann von der Einsatzumgebung automatisch ausgewertet werden können. Die Erstellung dieser Beschreibung und deren Auswertung in der Einsatzumgebung lassen sich wiederum werkzeugunterstützt durchführen.

  (Beispiel 4) Die Eigenschaften und Anforderungen von Softwarekomponenten können in separaten Dokumenten erfaßt sein, die von Programmen wie *InstallShield* [InstallShield] ausgewertet werden. Diese Beschreibungen lassen sich automatisiert erstellen, beispielsweise unter Nutzung des Standards *Open Software Description* (*OSD*) [W3C OSD].

Zusätzlich zu den bereits analysierten direkten Übergängen ergeben sich weitere Übergänge, die sich für eine Werkzeugunterstützung eignen.

- *Von Entwurf zu Integration und Test*
  In der Testphase wird das Zusammenwirken, also die zielgerichtete Interaktion der Softwarekomponenten überprüft. Die Tests lassen sich automatisieren, indem ein weiteres Softwaresystem (das Testsystem) erstellt wird, das mit dem eigentlichen System kommuniziert und funktionale Abläufe bzw. nichtfunktionale Eigenschaften überprüft. Die Komponenten des Testsystems lassen sich - so wie die Systemkomponenten selbst - aus der Systemspezifikation ableiten, sofern diese die nötigen Informationen beinhaltet. Die Ableitung eines Testsystems kann werkzeugunterstützt erfolgen.

  (Beispiel 5) Aus einer Systemspezifikation können mittels Werkzeugen automatisiert Testspezifikationen hergeleitet werden. Konkret können aus TTCN-Beschreibungen (*Tree and Tabular Combined Notation*, [ITUT X.292]) ausführbare Testsysteme erstellt werden. Dieser Ansatz ist in [VSB+ 99] konkretisiert.

- *Von Entwurf zu Einsatz und Wartung*
  Wenn eine Spezifikation Informationen enthält, die die Einsatzbedingungen eines Softwaresystems betreffen, z.B. Informationen über die Konfigurationsmöglichkeiten von Softwarekomponenten, so lassen sich aus diesen Informationen werkzeugunterstützt Beschreibungen erzeugen, die in der Einsatzphase verarbeitet werden können.

  (Beispiel 6) Die in *Beispiel 4* dargestellten Beschreibungen für den Übergang von der Integrations- in die Einsatzphase können entweder manuell erstellt oder aber automatisch aus der Spezifikation gewonnen werden, falls die Eigenschaften von Softwarekomponenten in der Spezifikation des Softwaresystems erfaßt sind. Einen derartigen Ansatz verfolgt das EURESCOM Projekt P924 "*Distribution and Configuration Support for Distributed PNO Applications*" [EU P924].

Bezieht man die automatisierbaren Übergänge in das Phasenmodell zur Softwareentwicklung ein, so ergeben sich die Zusammenhänge entsprechend Abb. 3.

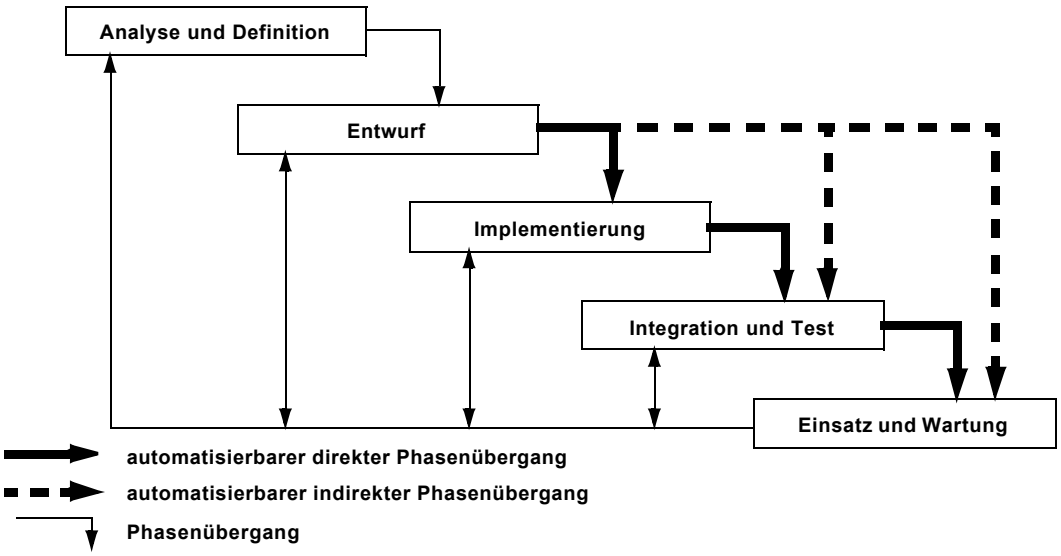


Abb. 3 Automatisierbare Übergänge im Phasenmodell mit Iterationen

Aus den vorangegangenen Darstellungen ergibt sich die zentrale Rolle der Entwurfsphase und deren Ergebnisse bezüglich der Möglichkeiten von automatisierten Übergängen. Sollen konkrete Entwicklungstechniken und Werkzeuge für die Unterstützung des Softwareentwicklungsprozesses bereitgestellt werden, so ergeben sich aus den obigen Betrachtungen folgende Hauptfragestellungen:

1. Wie werden Spezifikationen erstellt, die alle notwendigen Informationen für die automatisierbaren Übergänge enthalten?
2. Wie können diese Spezifikationen als Resultat der Entwurfsphase automatisiert in die Implementierungsphase abgebildet werden?
3. Wie können die aus der Implementierungsphase resultierenden Softwarekomponenten miteinander integriert werden?

In dieser Arbeit werden Antworten auf alle drei Fragestellungen gegeben und die zugehörigen Entwicklungstechniken unter Berücksichtigung der Anwendungsdomäne Telekommunikation detailliert. Es wird gezeigt, daß die objekt-orientierte Modellierung eine geeignete Antwort auf Frage 1 ist, die Aufstellung von Regeln für die automatisierte Ableitung von Softwarekomponenten aus Modellen die Antwort auf Frage 2 und der Einsatz von Komponentenarchitekturen die Antwort auf Frage 3.

Einen Beleg für die Aktualität dieser Fragestellungen liefert eine Strategiestudie (*Strategic Analysis Report*) von *Gartner Group*. So ist in 64% der befragten Unternehmen der Einsatz objektorientierter Entwicklungstechniken in der Entwurfs- und Implementierungsphase geplant bzw. bereits realisiert [Gart99a]. Darüber hinaus werden in mehr als 80% dieser Unternehmen Komponentenarchitekturen wie COM (*Component Object Model* [MSCOM]) oder CORBA bis einschließlich 2001 zur Unterstützung der Integrations- und Testphase eingeführt. Grundprinzip von Komponentenarchitekturen ist es, das Zusammenwirken (und somit die Integration) der Softwarekomponenten zu ermöglichen, falls bei deren Entwurf und Implementierung architekturspezifische Anforderungen (z.B. die Verwendung bestimmter Programmiersprachen oder die Einhaltung spezieller Schnittstellen) beachtet werden. Komponentenarchitekturen eignen sich insbesondere im Falle von verteilten Softwaresystemen, da die Heterogenität der Entwurfs-, Implementierungs- und Integrationsumgebung aber auch der Einsatzumgebung berücksichtigt werden muß.

Es sind derzeit noch keine integrierten Entwicklungstechniken verfügbar, die alle oben aufgeführten Fragestellungen für die Entwicklung von verteilten Telekommunikationssystemen präzise beantwortet. Die Aufgabe, der sich die Autoren mit dieser Arbeit stellen, besteht gerade in der technologischen Integration von objektorientiertem, modellbasiertem Entwurf und komponentenbasierter Implementierung und Integration solcher Softwaresysteme (vgl. Abb. 4).

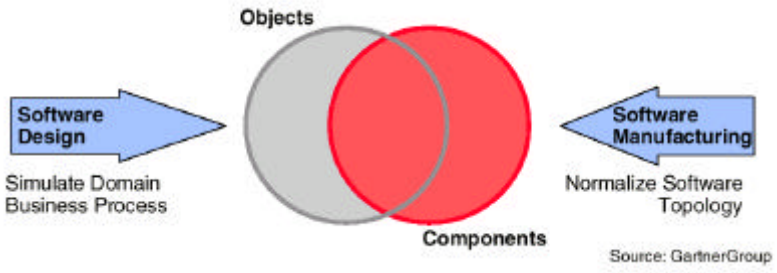


Abb. 4 Komponenten- und Objekttechnologie

Zunächst wird eine Einführung in Grundbegriffe gegeben, die sich mit den untersuchten Phasen Entwurf, Implementierung, Integration/Test und Einsatz/Wartung sowie der Modellierung verbinden. Diese Grundbegriffe sind Objekt und Interface (Entwurfsphase), Artefakt (Implementierungsphase), Softwarekomponente (nachfolgende Phasen) sowie Modellklasse und Konzeptraum (Modellierung). Anschließend wird

untersucht, welche Anforderungen an die Entwicklung von Telekommunikationssystemen und damit an unterstützende Entwicklungstechniken aktuell gestellt werden. Die Identifizierung dieser Anforderungen dient der Bewertung von bereits existierenden Entwicklungstechniken als auch des in dieser Arbeit vorgestellten Ansatzes.

1.1 Modellierung und Konzeptraum

Die vorangegangenen Diskussionen machen deutlich, daß das Ergebnis der Entwurfsphase - die Spezifikation des Softwaresystems - Voraussetzung für die Realisierung der Softwarekomponenten ist, aus denen das Softwaresystem letztendlich besteht. Insofern stellt sich die Frage nach den Inhalten der Spezifikation des Softwaresystems und den Entwicklungstechniken, die eingesetzt werden können, um eine möglichst präzise Spezifikation des Softwaresystems zu unterstützen.

In den letzten Jahren hat sich die objektorientierte Modellierung als *die* Entwicklungstechnik für den Entwurf verteilter Softwaresysteme etabliert. Dementsprechend bilden eine Menge von objektorientierten Modellen die Spezifikation des zu entwerfenden Softwaresystems. Modelle und Modellierung werden nicht nur innerhalb der Softwareentwicklung, sondern für jegliche Arten von Entwicklungsprozessen eingesetzt - ein Modell ist eine Abstraktion eines oder mehrerer Phänomene der objektiven Realität oder des menschlichen Denkens. Modelle sind der natürliche Weg, um diese Phänomene zu verstehen, zu untersuchen bzw. sie in die Realität zu überführen.

> *(Beispiel 7)* Die Eigenschaften physikalischer Entitäten werden mittels Modellen ermittelt, bevor diese in die Realität überführt werden: "*Lange bevor ein Flugzeug in der Realität fertig ist, existiert es in Form von Hunderten von Gigabyte an Daten auf den Rechnern der Airbus-Partner: numerische Strömungssimulatoren, Meßwerte aus dem Windkanal, Konstruktionszeichnungen für das Flugzeug selbst, für Systeme und Subsysteme, Pläne für die Verkabelung [...]*" [DBAG 98].

Offensichtlich werden eine Vielzahl von Modellen zur Untersuchung eines Systems eingesetzt, die jeweils auf bestimmte Aspekte des Systems fokussieren. Übertragen auf die Softwareentwicklung ergibt sich ein ähnliches Bild - es werden eine Vielzahl von Modellen eines zu entwickelnden Softwaresystems entworfen, wobei jeweils bestimmte Aspekte dieses Systems untersucht werden. Diese Modelle können dazu verwendet werden, die untersuchten Aspekte in die Realität, d.h. in die Systemkomponenten eines Softwaresystems zu überführen. Da komplexe Softwaresysteme ausschließlich werkzeugunterstützt entwickelt werden können, liegen die Modelle dieser Systeme in maschinell verarbeitbarer Form vor.

Es ist nun zu untersuchen, wie Modelle entstehen (Modellbildung). Phänomene des Denkens oder der Realität und deren Beziehungen (betrachtete Systemaspekte) werden durch Abstraktion in Modelle überführt. Diese Abstraktion erfolgt auf der Grundlage von Modellkonzepten und -beziehungen, die die wesentlichen Eigenschaften einer Gruppe von Phänomenen beschreiben und von Unterschieden abstrahieren, die als unwesentlich erachtet werden. Modelle selbst beinhalten somit Instanzen dieser Modellierungskonzepte und -beziehungen, die die als wesentlich erachteten Eigenschaften des betrachteten Phänomens konkretisieren. Die Modellierungskonzepte und -beziehungen werden in der objektorientierten Modellierung durch Anwendung allgemeiner objektorientierter Prinzipien wie Klassifizierung, Instanziierung, Generalisierung, Spezialisierung, Komposition, Dekomposition und Assoziation definiert. Die Gesamtheit aller zur Modellbildung verwendbaren Modellierungskonzepte und -beziehungen werden für eine betrachtete Domäne zu einem spezifischen Konzeptraum zusammengefaßt. Die in dieser Arbeit betrachtete Domäne ist die Entwicklung von verteilten Telekommunikationssoftwaresystemen.

Da Konzepträume sehr komplex sein können, wird häufig eine Systematisierung von Modellierungskonzepten und -beziehungen vorgenommen. Diese führt zu Modellklassen, denen jeweils spezielle Funktionen im Entwurfsprozeß zugeordnet werden können. Eine derartige Systematisierung für die Entwicklung verteilter Softwaresysteme und damit die Definition von Modellklassen und deren Funktionen erfolgte u.a. im Referenzmodell für offene, verteilte Verarbeitung (*Reference Model for Open Distributed Processing*

[ITUT X.902],[ITUT X.903] und [ITUT X.904]) sowie in der Definition des *Unified-Modelling-Language*-Standards [OMGUML1.3]. Beide Ansätze werden bezüglich der Anforderungen diskutiert, die die Domäne Telekommunikation impliziert.

Die Definition eines Konzeptraumes für die Entwicklung von verteilten Telekommunikationssoftwaresystemen ist fundamental. Allerdings stellt sich die Frage, auf welche Weise Modelle, die auf diesem Konzeptraum basieren, von einem Entwickler definiert und zwischen Entwicklungswerkzeugen ausgetauscht werden. Zu diesen Zwecken werden eine oder mehrere Notationen benutzt, die eine Visualisierung der Instanzen der Konzepte des Konzeptraumes und deren Beziehungen definieren.

(Beispiel 8) Die Komponentenarchitektur CORBA definiert einen Konzeptraum zur Beschreibung von Schnittstellen von Softwarekomponenten. Die Beschreibungssprache CORBA-IDL (*Interface Definition Language*) ist eine Notation für diesen Konzeptraum.

1.2 Softwarekomponente

Der Terminus Softwarekomponente eines verteilten Softwaresystems ist grundlegend für das Verständnis der Entwicklungsphasen Implementierung, Integration/Test und Einsatz/Wartung. In der Literatur werden mit diesem Begriff leider Konzepte mit beträchtlicher semantischer Diskrepanz adressiert (z.B. Softwarekomponentendefinition in [Szy 99] vs. Metatypdefinition *Component* in [OMG CCM I]), so daß eine Präzisierung im Kontext dieser Arbeit erforderlich ist.

Allen Darstellungen gemeinsam ist die prinzipielle Vorstellung, Softwarekomponenten als physisch repräsentierte, kombinierbare Einheiten zu betrachten, die Instruktionssequenzen (Codemodule) beinhalten. Diese Sequenzen lassen sich maschinell ausführen. Dazu muß eine physikalische Repräsentation der Softwarekomponente (z.B. in Form einer Datei) auf der ausführenden Maschine bereitgestellt werden. Zusätzlich zu der Bereitstellung ist ggf. eine Installation erforderlich, die durch das Setzen von Parametern, wie z.B. Umgebungsvariablen, die Maschine in die Lage versetzt, den zuvor bereitgestellten Repräsentanten der Softwarekomponente tatsächlich auszuführen. Bereitstellung und Installation von Softwarekomponenten wird in der Literatur auch als *Deployment* bezeichnet [OMG CCM I]. Softwarekomponenten können also als *Deployment*-Einheiten (*Deployment Units*) betrachtet werden. Falls mehrere Softwarekomponenten gemeinsam

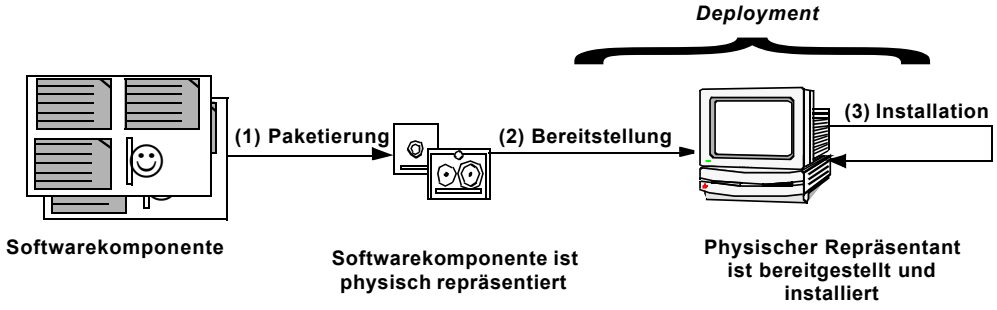


Abb. 5 Softwarekomponenten und Deployment

bereitgestellt werden sollen (z.B. wenn ein System oder Teilsystem aus mehr als einer Komponente besteht), werden sie vorher zu Paketen zusammengefaßt, wobei dann das entstehende Paket als singuläre *Deployment*-Einheit bereitgestellt und installiert wird (vgl. Abb. 5).

Szyperski definiert, daß Softwarekomponenten wohldefinierte Schnittstellen und ausschließlich explizite - d.h. vollständig beschriebene - Umgebungsabhängigkeiten besitzen [Szy 99]. Aufgrund dieser wohldefinierten Schnittstellen können Softwarekomponenten zu komplexen Strukturen zusammengesetzt werden, die

ihrerseits wiederum Softwarekomponenten sein können (z.B. Erstellung einer Softwarebibliothek aus bereits vorhandenen Bibliotheksmodulen). Szyperski präzisiert weiterhin, daß Softwarekomponenten nicht nur Instruktionssequenzen (Codemodule, z.B. Menge von Klassendeklarationen und -implementierungen einer objektorientierten Programmiersprache oder Menge von Funktionen/Prozeduren einer nicht objektorientierten Programmiersprache) beinhalten, sondern daß auch durch diese Codemodule referenzierte Ressourcen (z.B. elektronische Bilder), die nicht der Codegenerierung eines Compilers entstammen, Bestandteil von Softwarekomponenten sein können. Dieser Erweiterung der initialen Definition einer Softwarekomponente wird in der vorliegenden Arbeit gefolgt.

Weiterhin ist allgemein anerkannt, daß Softwarekomponenten einerseits Kompositionen von Codemodulen und Ressourcen darstellen, andererseits der Zweck einer Softwarekomponente wiederum Komposition ist [ITUT X.902][ITUT X.903][OMG CCM I]. Die Ansichten bezüglich der Eigenschaften der in einer Softwarekomponente enthaltenen Codemodule als auch bezüglich der Charakteristika der Softwarekomponentenschnittstellen variieren jedoch stark. Um eine Präzisierung zu erreichen, ist es notwendig, diejenigen Elemente einer Spezifikation zu definieren, die durch die Codemodule, die in Softwarekomponenten enthalten sind, realisiert werden. Da bei Anwendung der in dieser Arbeit vorgestellten Entwicklungstechniken der Entwurf von Softwarekomponenten durch objektorientierte Modellierung erfolgt und die Softwarekomponenten aus den resultierenden Modellen automatisch abgeleitet werden, sind diese Elemente gerade Objekte.

1.3 Objekt und Interface

Der Terminus Objekt ist grundlegend für die objektorientierte *Modellierung* verteilter Softwaresysteme, impliziert jedoch nicht zwingend eine objektorientierte *Implementierung* von Systemkomponenten.

Ein Objekt ist das Modell einer Entität. Eine Entität bezeichnet dabei jedes konkrete oder abstrakte Phänomen von Interesse in der betrachteten Domäne. Ein Objekt ist unterscheidbar von allen anderen Objekten (Identität), sowie charakterisiert durch sein Verhalten und seinen Zustand. Objekte kapseln Zustand und Verhalten, Zustandsänderungen können ausschließlich als Resultat interner Aktionen (Aktivität ohne Beteiligung der Objektumgebung) oder Interaktionen (Aktivität unter Beteiligung der Objektumgebung) auftreten [ITUT X.902][Szy 99]. Die Objektumgebung ist gerade der Teil des verteilten Softwaresystems, der nicht Bestandteil des Objektes ist.

Die funktionale Dekomposition des verteilten Softwaresystems in Objekte und die Definition des Zusammenwirkens zwischen diesen potentiell verteilbaren Objekten sind grundlegende Aufgaben der objektorientierten Modellierung in der Entwurfsphase. Eine Modellklasse, die auf diese Aspekte fokussiert, wird bereits im Referenzmodell für offene und verteilte Verarbeitung (RM-ODP) in [ITUT X.903] definiert. Die in [ITUT X.903] definierten Konzepte der *Computational*-Modellklasse erlauben die Verteilung eines Systems durch funktionale Dekomposition des Systems in via Interfaces interagierende *Computational*-Objekte (COs). Das Interface eines COs ist eine referenzierbare Zusammenfassung von potentiellen Interaktionen dieses COs. Die Interaktion zwischen COs erfolgt auf der Grundlage dreier Interaktionsmodelle [ITUT X.903]:

- Unidirektionale Kommunikation atomarer, endlicher Nachrichten (Signalinteraktion),
- Unidirektionale Kommunikation kontinuierlicher, potentiell unendlicher Datenströme (*Continuous-Media*-Interaktion) sowie
- Aufruf einer Operation an einem entfernten Objekt (operationale Interaktion).

Weiterhin wird das Konzept von Typen von *Computational*-Objekten im Sinne einer zur Instanziierung geeigneten Beschreibung definiert (CO-Typ). Den im RM-ODP vorgenommenen Definitionen von CO, CO-Typ und Interaktionsarten wird in dieser Arbeit gefolgt[1].

1. Typ wird hier statt des mißverständlichen Terminus *Template* verwendet.

1.4 Artefakt

Eine Entwicklungstechnik, deren Ziel in der automatischen Ableitung der Softwarekomponenten aus Modellen besteht, muß definieren, auf welche Weise die Konzepte der verwendeten Modellklassen auf Softwarekomponenten abgebildet werden. Die Konzepte CO-Typ und Interface werden durch spezifische programmiersprachliche Konstrukte repräsentiert, deren Instanzen gerade das Verhalten, die Identität und den Zustand von COs realisieren. Diese programmiersprachlichen Konstrukte, deren Abstraktionen als *Artefakte*[1] bezeichnet sind, werden zu Codemodulen zusammengefaßt, die in Softwarekomponenten enthalten sind. Der Terminus Artefakt abstrahiert von einer konkreten programmiersprachlichen Realisierung, wie z.B. Klasse einer objektorientierten Programmiersprache, um die hier vorgestellten Entwicklungstechniken unabhängig von den Konzepten verschiedener Programmiersprachen definieren zu können. Somit enthalten Softwarekomponenten programmiersprachliche Konstrukte (modelliert durch Artefakte), deren Instanzen (bzw. Teilmengen aller dieser Instanzen) den Zustand und die Identität von COs repräsentieren und deren Verhalten realisieren. Derartige Instanzen werden während der maschinellen Verarbeitung einer Softwarekomponente erzeugt. Ein CO korrespondiert i.allg. nicht ausschließlich mit einer einzigen solchen Instanz. Die Identität von COs und deren Verhalten wird durch eine Menge von Instanzen realisiert, die durch Komponentenarchitekturen unterstützt werden.

(Beispiel 9) Sei *Calculator* ein CO-Typ, dessen Instanzen die Funktionalität eines wissenschaftlichen Taschenrechners der Umgebung bereitstellen. Dann kann dieser CO-Typ durch Benutzung von Klassen einer mathematischen Softwarebibliothek realisiert werden. Diese Klassen werden im Kontext des Entwurfsmodells des Taschenrechners durch Artefakte modelliert.

Komponentenarchitekturen offerieren neben Kommunikationsmechanismen auch Mechanismen zur Repräsentation der Identität von COs. In konkreten Repräsentationen von Artefakten ist auf dieser Basis das gewünschte Verhalten von CO-Typen ("*Business Logic*") implementiert (vgl. Abb. 6 in UML-Notation).

1.5 Entwicklung verteilter Telekommunikationssoftwaresysteme - Anforderungen

Um die Anforderungen an die Entwicklung von verteilten Softwaresystemen im Telekommunikationskontext unterstützende Entwicklungswerkzeuge konkretisieren zu können, sind zunächst die Charakteristika von Telekommunikationssoftwaresystemen zu untersuchen: Durch welche Eigenschaften zeichnen sich verteilte Telekommunikationssoftwaresysteme aus und welche Anforderungen an Entwicklungstechniken, die den Entwurf und die Realisierung derartiger Systeme unterstützen, lassen sich daraus ableiten?

- *Offenheit*

  Offenheit ist eine Kerneigenschaft verteilter Softwaresysteme im Telekommunikationskontext. Sie umfaßt sowohl die Eigenschaft der Portabilität als auch die Sicherung der Kooperationsfähigkeit von Komponenten eines Softwaresystems. Die Portabilität von Softwarekomponenten sichert die Ausführbarkeit dieser Komponenten unter unterschiedlichen Einsatzbedingungen ohne Modifikationen. Es wird zwischen binärportablen und quellenportablen Softwarekomponenten unterschieden. Binäre Portabilität erlaubt die Portierung von Softwarekomponenten selbst, während Quellenportabilität die automatische Erstellung einer Softwarekomponente aus der zu portierenden Softwarekomponente ohne manuelle Modifikation gestattet. Kooperation von Softwarekomponenten bedeutet zielgerichtete Interaktionen zwischen u.U. räumlich verteilten Softwarekomponenten. Zielgerichtete Interaktionen seien im Sinne der Interoperabilität verstanden: Systemkomponenten müssen in der Lage sein, auf gewünschte, in der Spezifikation des Softwaresystems definierte Art und Weise zusammenzuarbeiten. Kooperation bedeutet also

1. Artefakt ist die deutsche Übersetzung für den in diesem Kontext (vgl. [OMGCCM I]) gebräuchlichen englischen Terminus *Artifact.*

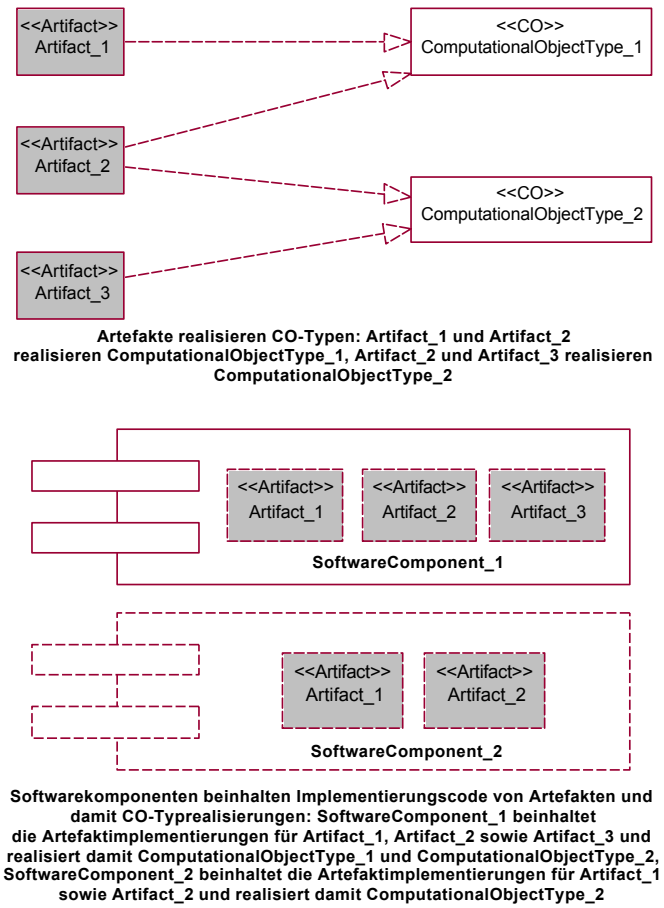


Abb. 6 CO-Typ, Artefakt und Softwarekomponente

insbesondere die Offenlegung der Schnittstellen der Systemkomponenten sowie die Definition des an diesen Schnittstellen zu erwartenden Verhaltens. Ein verteiltes Softwaresystem, das die Eigenschaft der Offenheit besitzt, wird als *offenes* verteiltes System bezeichnet.

Entwicklungswerkzeuge, die den Entwurf, die Implementierung und die Integration von verteilten Softwaresystemen im Telekommunikationskontext unterstützen, müssen einerseits die Definition der Schnittstellen des Softwaresystems und des an diesen zu erwartenden Verhaltens erlauben. Andererseits muß es möglich sein, mit diesen Werkzeugen mindestens quellenportable Softwarekomponenten zu produzieren, die portiert werden können. Darüber hinaus sollten solche Werkzeuge Mechanismen realisieren, die alle Informationen der Spezifikation von Softwarekomponenten in den Phasen Implementierung, Integration/Test und Einsatz/Wartung zur Verfügung stellen.

- *Konzeptionelle Offenheit*

  Die oben beschriebene Offenheitsanforderung bezieht sich auf die unter Anwendung von Entwicklungstechniken realisierten Systeme. Da die technologische Entwicklung gerade im Telekommunikationssektor aber immens schnell voranschreitet, ist nicht nur die Offenheit der Telekommunikationssysteme selbst, sondern auch die konzeptionelle Offenheit der Entwicklungstechniken zum Entwurf, zur Implementierung und zur Integration gefordert. Neue technische Möglichkeiten oder Anforderungen an Telekommu-

nikationssysteme sollen sich in Form von geeigneten Konzepten in die Entwicklungstechniken integrieren lassen und nicht deren Neuentwicklung erfordern.

Zur Realisierung der konzeptionellen Offenheit müssen bei der Definition der Entwicklungstechniken geeignete Mechanismen verwendet werden, die die spätere flexible Erweiterung dieser Techniken hinsichtlich zusätzlicher zu unterstützender Konzepte oder Technologien gestattet.

- *Gütebeschreibung und Garantie*

  Der Systemzweck von verteilten Telekommunikationssoftwaresystemen besteht i.allg. in der Bereitstellung *abrechenbarer* Leistungen. Die Grundlage der Abrechnung ist das garantierte Erbringen dieser Leistungen mit einer bestimmten Güte. Während des Entwurfs und der Realisierung (d.h. der Implementierung, der Integration und der Bereitstellung) verteilter Telekommunikationssoftwaresysteme müssen Mechanismen unterstützt werden, die die Sicherstellung der Leistungserbringung mit einer zugesagten Qualität gewährleisten. Die Sicherstellung des Zugriffs auf bestimmte Funktionalität innerhalb festgelegter Zeitspannen sowie die Sicherheit von Transaktionen sind hier Beispiele.

  Entwicklungswerkzeuge, die den Entwurf und die Realisierung von Telekommunikationssoftwaresystemen unterstützen, müssen geeignete Ausdrucksmittel im Kontext des Entwurfs sowie geeignete Ableitungsregeln bereitstellen.

- *Flexible Skalierbarkeit*

  Verteilte Telekommunikationssoftwaresysteme erbringen Leistungen für eine sehr große Zahl von Anwendern. Dabei muß zwischen der sehr großen Anzahl *potentieller* Anwender und der i.allg. stark variierenden Anzahl der das System *gleichzeitig* benutzenden Anwender unterschieden werden. Eine hohe Anzahl das System gleichzeitig benutzender bzw. potentieller Anwender soll bis zu einer bestimmten Grenze keine, oder zumindest tolerierbare Auswirkungen auf die Leistungsfähigkeit des verteilten Telekommunikationssystems haben (Skalierbarkeit).

  Skalierbarkeit ist dabei im Sinne von Flexibilität zu verstehen: Während des Entwurfs und der Realisierung verteilter Telekommunikationssoftwaresysteme müssen Mechanismen unterstützt werden, mit deren Hilfe flexibel auf die sich in kurzen Zeiträumen verändernden Benutzungssituationen reagiert werden kann. Entwicklungstechniken, die den Entwurf und die Realisierung solcher Systeme unterstützen, müssen geeignete Ausdrucksmittel sowie Ableitungsregeln für diese Mechanismen bereitstellen.

- *Flexible Adaptierbarkeit*

  Beim Einsatz von Telekommunikationssystemen können Situationen erkannt werden, die eine Veränderung der Spezifikation des Softwaresystems notwendig machen. Diese Situationen sind im Vorfeld des Systemeinsatzes nicht bekannt und können daher im Entwurf nicht erfaßt werden.

  Entwurf und Realisierung verteilter Telekommunikationssysteme unterstützende Entwicklungstechniken müssen eine flexible Weiterentwicklung des Softwaresystems gewährleisten, da diese veränderten Anforderungen sehr schnell berücksichtigt werden müssen. Insbesondere ist eine Nachnutzbarkeit der aus der Entwurfsphase resultierenden Spezifikationen anzustreben.

- *Unterstützung von Continuous-Media-Interaktionen*

  Verteilte Telekommunikationssoftwaresysteme erfordern oft die kontinuierliche Übertragung von Medien (Audio, Video, Daten) zwischen u.U. räumlich verteilten Maschinen.

  Entwicklungstechniken zur Unterstützung des Entwurfs und der Realisierung solcher Systeme müssen neben der Beschreibung der für die Übertragung geeigneten Mechanismen auch die Integration dieser Interaktionen mit anderen Interaktionsarten ermöglichen.

- *Flexible Integration von Softwarekomponenten*

  Der Einsatz von verteilten Telekommunikationssoftwaresystemen erfolgt i.allg. unternehmensübergreifend, d.h. an dem Betrieb eines solchen Systems sind nicht mehr nur die traditionellen Telekommunikati-

onsnetzbetreiber beteiligt, sondern eine Vielzahl anderer Unternehmen. Naturgemäß sind dabei die verwendeten Systemkomponenten heterogen, d.h. unter Benutzung verschiedener Technologien realisiert.

Entwicklungstechniken zur Unterstützung des Entwurfs und der Realisierung dieser Komponenten müssen einerseits die Heterogenität der in Telekommunikationsnetzen verwendeten Softwarekomponenten berücksichtigen und andererseits Mechanismen bereitstellen, die die Integration dieser Softwarekomponenten ermöglichen.

- *Kurze Entwicklungszeiten*

  Durch den hohen Konkurrenzdruck im Telekommunikationsmarkt muß der Entwicklungszeitraum für verteilte Telekommunikationssoftwaresysteme - verglichen mit deren Einsatzzeitraum - sehr klein sein. Da der Einsatzzeitraum bei derartigen Systemen zunehmend kleiner wird, gewinnt diese Forderung an Bedeutung.

  Entwicklungstechniken müssen Mechanismen bereitstellen, die diese kurzen Entwicklungszeiträume ermöglichen. Dazu zählt die weitgehende Unterstützung der Wiederverwendbarkeit von existierenden Softwarekomponenten dieser Systeme sowie deren Spezifikationen.

Eine grundlegende Anforderung an Entwicklungstechniken zur Unterstützung der Entwicklung jeglicher Art von Softwaresystemen ist die Bereitstellung von Entwicklungswerkzeugen, die diese Techniken automatisieren - „*Die Tatsache, ob eine Werkzeugunterstützung für ein Vorgehensmodell bzw. eine Methode existiert oder nicht, entscheidet fast schon allein über deren Einsatz. Mehr noch, ohne Werkzeugunterstützung ist die Entwicklung komplexer Software weder planbar noch kosten- und termingerecht durchführbar*“[1].

Die Erfahrung in der Softwaretechnik gebietet sogar noch eine Verschärfung der obigen Aussage. Die Werkzeugunterstützung muß professionell gerfolgen. Die Bindung an Entwicklungstechniken und -werkzeuge zur Unterstützung der Softwareentwicklung ist eine existenzielle Entscheidung für Unternehmen, die Softwarekomponenten entwickeln. Fehlentscheidungen können hier unternehmensbedrohende Folgen haben, entsprechend hohe Anforderungen werden daher an Werkzeuge gestellt und vor dem Einsatz konkreter Werkzeuge überprüft.

Der oben aufgeführte Anforderungskatalog faßt die Erfahrungen der Autoren aus der Durchführung einer Vielzahl von Softwareentwicklungsprojekten im Telekommunikationsbereich zusammen. Eine Aussage über die Vollständigkeit des Katalogs ist aufgrund der Breite der Anwendungsdomäne schwierig, wenn nicht gar unmöglich. Allerdings lassen sich die einzelnen Anforderungen durch Studien belegen, die von unabhängigen Organisationen angefertigt wurden. Exemplarisch sei hier auf [Gart99a] verwiesen.

Viele der beschriebenen Eigenschaften von verteilten Telekommunikationssoftwaresystemen sind auch auf andere Anwendungsdomänen übertragbar und insofern nicht telekommunikationsspezifisch. Jedoch ist die Zusammenfassung dieser Eigenschaften gerade spezifisch für Telekommunikationssoftwaresysteme.

1.6 Entwicklungstechniken im Entwicklungsprozeß

Konkrete, in der Praxis eingesetzte Softwareentwicklungsprozesse basieren prinzipiell auf dem Phasenmodell, jedoch verfeinern und erweitern sie dieses. Solche Softwareentwicklungsprozesse werden in Kapitel2 vorgestellt. Das grundsätzliche Ziel dieser Arbeit besteht in der Bereitstellung von Entwicklungstechniken, die sich in diese Softwareentwicklungsprozesse einordnen lassen und dabei gerade diejenigen Phasen und Phasenübergänge unterstützen, die als zeitaufwendig identifiziert und darüber hinaus als automatisierbar erachtet werden. Automatisierung bedeutet die Realisierung von Entwicklungswerkzeugen, die diese Entwicklungstechniken unterstützen.

1. [PS94] Seite 60

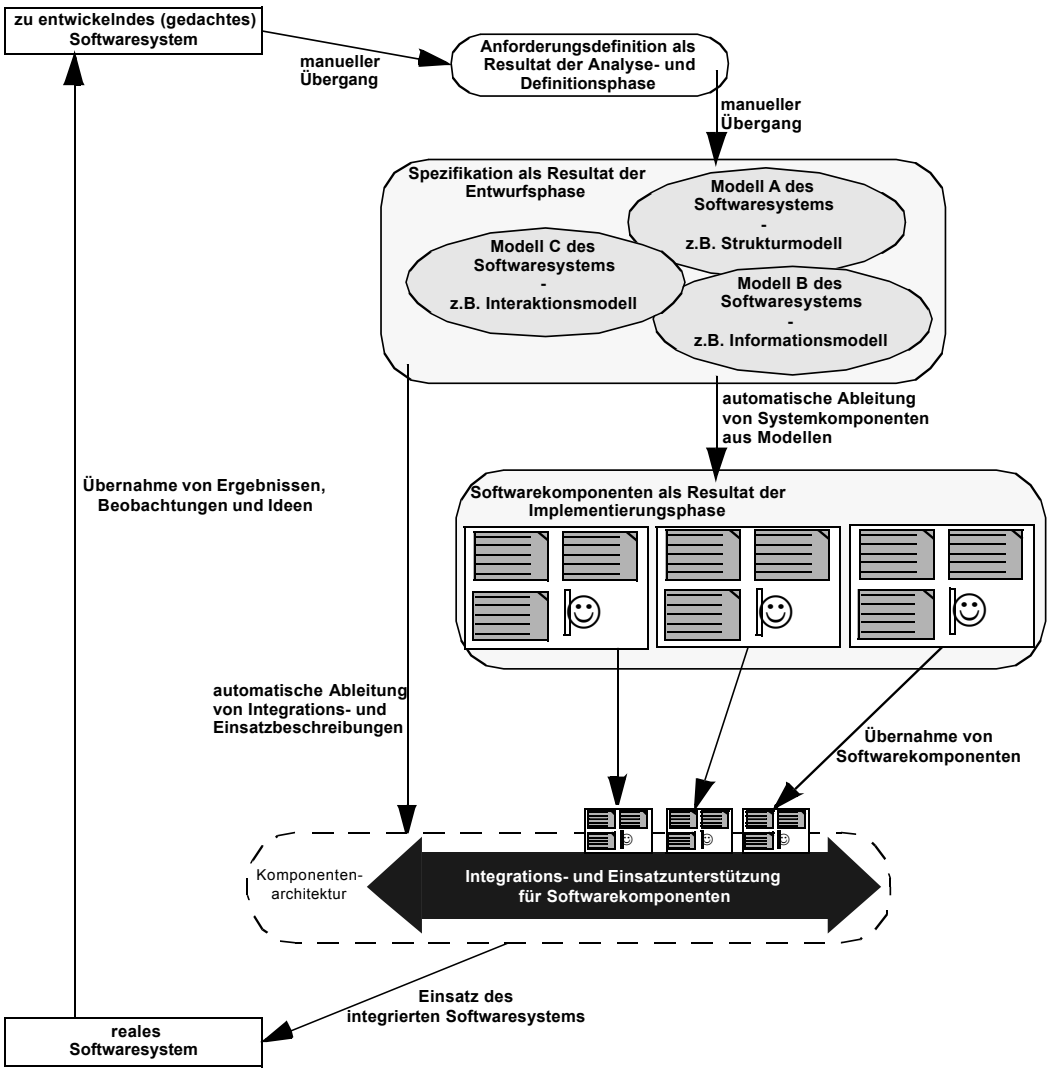


Abb. 7 Entwicklung von Softwaresystemen

Es kann festgestellt werden, daß sich objektorientierte Modellierung als zentrale Entwicklungstechnik für die Entwurfsphase durchgesetzt hat. Der Diskussion in Abschnitt 1.1 folgend, ist die Definition eines Konzeptraumes für die Entwicklung von verteilten Telekommunikationssoftwaresystemen grundlegend. Der Nutzen des Einsatzes der objektorientierten Modellierung ist dann besonders hoch, falls mittels der Entwicklungstechnik automatische Ableitung von Softwarekomponenten die Spezifikation als Resultat der Entwurfsphase automatisch in die Implementierungs- und Integrationsphase abgebildet werden kann. Die Definition des Konzeptraumes ist nicht nur zentrale Voraussetzung für die objektorientierte Modellierung, sondern bildet auch die Grundlage für die Definition von Ableitungsregeln, die in ihrer Gesamtheit gerade die Entwicklungstechnik automatische Ableitung von Softwarekomponenten ergeben. Entsprechend den Diskussionen in Abschnitt 1.3 sind CO-Typ, CO und Interface zentrale Konzepte eines Konzeptraumes für verteilte Telekommunikationssoftwaresysteme. Um eine Integration der objektorientierten Modellierung mit der automatischen Ableitung von Softwarekomponenten zu ermöglichen, ist darüber hinaus die Aufnahme der Konzepte Softwarekomponente und Artefakt mit ihren wesentlichen Eigenschaften in einen solchen Konzeptraum geboten, um die Realisierungsbeziehungen zwischen CO-Typen und den diese realisierenden Codemodulen in Softwarekomponenten abzubilden.

In der Integrationsphase werden die aus der Implementierungsphase resultierenden Softwarekomponenten miteinander integriert, und damit das zu entwickelnde Softwaresystem realisiert. Diese Integration erfolgt unter Benutzung von Komponentenarchitekturen. Unter der Voraussetzung, daß die zu integrierenden Softwarekomponenten und deren Eigenschaften bereits in der Entwurfsphase in einem Modell reflektiert sind, können aus der Spezifikation, die dieses Modell enthält, detaillierte Konfigurationsbeschreibungen ebenfalls automatisch abgeleitet werden. Damit läßt sich der Aufwand für die Integrationsphase verringern.

Die Einordnung der Entwicklungstechniken objektorientierte Modellierung, automatische Ableitung von Softwarekomponenten und Einsatz einer Komponentenarchitektur in Entwicklungsprozesse, die auf dem Phasenmodell basieren, veranschaulicht Abb. 7.

Dieser Diskussion folgend lassen sich die Ziele dieser Arbeit konkretisieren:

- es ist zu untersuchen, auf welche Weise sich die identifizierten Entwicklungstechniken in praxisrelevante Softwareentwicklungsprozesse integrieren lassen (vgl. Kapitel2),
- es ist auf der Basis existierender Ansätze eine präzise Definition des Konzeptraumes vorzunehmen, der geeignet ist, die in Abschnitt 1.5 diskutierten Anforderungen zu erfüllen (vgl. [CoRE II], Kapitel3),
- es sind Notationen zu definieren, die die Präsentation von Entwurfsmodellen unterstützen (vgl. [CoRE II], Kapitel5),
- es ist eine Komponentenarchitektur zu präzisieren, die geeignet ist, die in Abschnitt 1.5 diskutierten Anforderungen hinsichtlich der Integration von Softwarekomponenten zu erfüllen (vgl. [CoRE III], Kapitel1),
- es sind automatisierbare Ableitungsregeln zu formulieren, die eine Spezifikation als Resultat des Entwurfsprozesses in die nachfolgenden Phasen überführen (vgl. [CoRE III], Kapitel 2) und
- es sind Werkzeuge zu realisieren, die die Entwicklungstechniken unterstützen (vgl. [BK 01a]).

KAPITEL 2

Ausgangslage

Die vorangegangenen Untersuchungen schlossen mit der These, daß der Einsatz der Entwicklungstechniken objektorientierte Modellierung, automatische Ableitung von Softwarekomponenten und Einsatz von Komponentenarchitekturen zum einen den zeitlichen Aufwand bei der Entwicklung verteilter Softwaresysteme reduziert und zum anderen die Qualität der resultierenden Softwaresysteme bezüglich der in Abschnitt 1.5 vorgestellten Anforderungen verbessert. Es wurde diskutiert, daß Softwaresysteme durch einen Softwareentwicklungsprozeß entstehen, in dem i.allg. Entwicklungstechniken eingesetzt werden. Aus diesen Thesen ergeben sich die folgenden zu untersuchenden Fragen:

- Welche praxisrelevanten Entwicklungsprozesse können identifiziert werden?
- Wie können die Entwicklungstechniken objektorientierte Modellierung, automatische Ableitung von Softwarekomponenten und Einsatz von Komponentenarchitekturen im Kontext dieser Entwicklungsprozesse eingesetzt werden?
- Gibt es existierende Ansätze für die Entwicklung verteilter Telekommunikationssoftwaresysteme, die bereits Aspekte dieser Entwicklungstechniken realisieren?

Die nachfolgenden Ausführungen geben Antworten auf diese Fragen.

2.1 Entwicklungsprozesse

In den vergangenen Jahren wurden Entwicklungsprozesse für Softwaresysteme im akademischen und industriellen Bereich vorgestellt, von denen einige eine weite Verbreitung erreicht haben. Es ist festzustellen, daß nicht alle Organisationen, die im Bereich der Softwareentwicklung tätig sind, die verwendeten Softwareentwicklungsprozesse publizieren. Es besteht jedoch die Forderung nach der Kooperation zwischen Unternehmen im Bereich der Softwareentwicklung, so daß mindestens eine Vereinheitlichung der verwendeten Terminologie in Softwareentwicklungsprozessen angestrebt wird. Als ein weit verbreiteter Ansatz ist *Rational Unified Process* (*RUP*, [RationalRUP]) herauszustellen, der in vielen Organisationen eingesetzt wird. Im

öffentlichen Bereich der Bundesrepublik Deutschland hat die große Zahl von beauftragten Organisationen, die Softwaresysteme entwickeln und betreiben, dazu geführt, daß ein vereinheitlichter Softwareentwicklungsprozeß - das V-Modell - *de facto* normiert wurde, der sich in vielen, vor allem deutschen Unternehmen durchgesetzt hat.

Diese drei Ansätze, *Rational Unified Process*, das V-Modell und die Vereinheitlichung der Terminologie von nicht öffentlich publizierten Softwareentwicklungsprozessen werden im folgenden untersucht. Dabei erfolgt eine Charakterisierung des jeweiligen Ansatzes und eine Diskussion der Einordnung der Entwicklungstechniken objektorientierte Modellierung, automatische Ableitung von Softwarekomponenten und Einsatz von Komponentenarchitekturen in den Softwareentwicklungsprozeß.

2.1.1 Rational Unified Process

2.1.1.1 Charakterisierung

RUP ist ein durch das Unternehmen *Rational Software Corp.* entwickelter Softwareentwicklungsprozeß, in den viele vorab existierende Prozesse einflossen. Entsprechend einer durch das Unternehmen erfolgten Untersuchung der Praktiken, die bei der Durchführung erfolgreicher Softwareentwicklungsprojekte in industriellen und anderen Organisationen Anwendung finden, konnten sechs derartige Praktiken als besonders wichtig bei der Entwicklung von Software herausgestellt werden.

- Iterative Softwareentwicklung

  Im Gegensatz zum rein sequentiellen Phasenmodell kann durch die Anwendung des iterativen Phasenmodells in RUP die Identifikation von Fehlern zu einem möglichst frühen Entwicklungszeitpunkt erreicht werden.

- *Verwaltung der Anforderungen*

  Bei der Entwicklung von Softwaresystemen kommt es oft während der Entwicklungszeit zu Änderungen der Anforderungen an das fertige Produkt - das Softwaresystem. Zudem ist es unmöglich, alle Anforderungen an ein Softwaresystem bereits in der Analyse- und Definitionsphase zu erkennen. Die Anforderungsverwaltung in RUP systematisiert und dokumentiert die Anforderungen an die Außenwirkung des Softwaresystems sowie alle diesbezüglichen Entscheidungen.

- *Modularisierung des zu entwickelnden Softwaresystems*

  Die Entwicklung von Softwaresystemen erfolgt in RUP mit dem Ziel einer modularen Architektur des resultierenden Softwaresystems. Dementsprechend können während der Realisierung der Softwaresysteme Komponentenarchitekturen zum Einsatz kommen, die die Integration der entstehenden Softwarekomponenten, deren Test und Überführung in den Einsatz erheblich erleichtern.

- *Visuelle Modellierung*

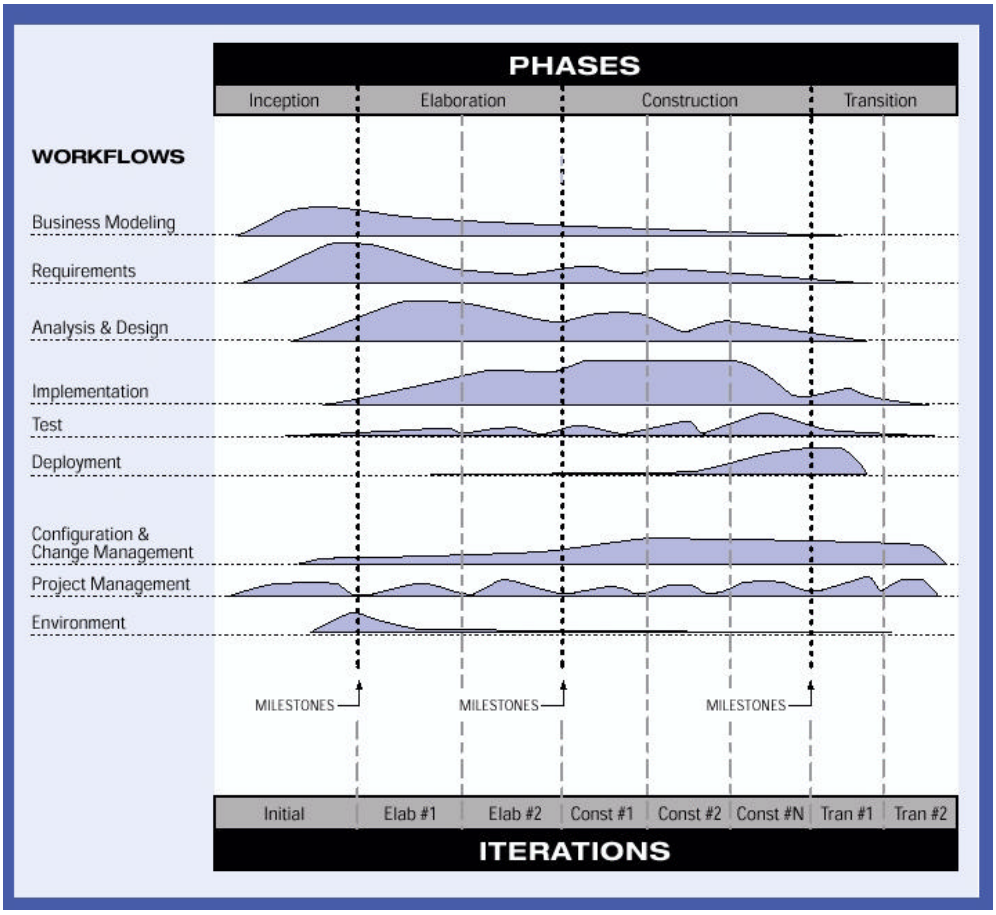
  RUP verwendet die objektorientierte Modellierung zu Entwurf von Softwaresystemen. Konkret wird in RUP die standardisierte Notation *Unified Modeling Language* [OMGUML1.3] zur Anwendung vorgeschlagen. Dadurch wird die Kommunikation von Spezifikationsteilen zwischen Entwicklern in standardisierter Art und Weise ermöglicht.

- *Verifikation der Qualität der Software*

  Die Qualität von Softwaresystemen wird während des mittels RUP durchgeführten Entwicklungsprozesses bezüglich funktionaler und nicht-funktionaler Eigenschaften durch die notwendige Erstellung von Testspezifikationen und deren Durchführung gesichert.

- *Kontrolle der Veränderungen an der Software*

  Ein fester Bestandteil von RUP ist die Nachvollziehbarkeit von Änderungen am Entwurf des Softwaresystems.



Source: Rational Software Corp.

Abb. 8 Phasen und *Work Flows* in RUP

Zur Unterstützung dieser Praktiken definiert RUP zeitliche Phasen (RUP-Phasen) eines Softwareentwicklungsprozesses, die mit den Phasen des Phasenmodells korrespondieren (vgl. Abb. 8[1]):

- In der *Inception*-Phase werden die Ziele des zu entwickelnden Softwaresystems fixiert und die wesentlichen Anforderungen an dieses definiert,
- in der *Elaboration*-Phase wird die Spezifikation des Softwaresystems erstellt,
- in der *Construction*-Phase wird das Softwaresystem implementiert, die entstehenden Softwarekomponenten werden integriert und
- in der *Transition*-Phase wird das Softwaresystem in den Produktionsbetrieb überführt und gewartet.

Die Besonderheit von RUP gegenüber dem Phasenmodell ist jedoch, daß innerhalb jeder RUP-Phase eine Menge von Iterationen erfolgt. Jede derartige Iteration ist wiederum entsprechend des Phasenmodells strukturiert, wobei die Bedeutung der einzelnen Phasen und deren zeitlicher Aufwand von der jeweiligen RUP-

1. Im Diagramm ist der zeitliche Aufwand von *Work Flows* in den einzelnen RUP-Phasen dargestellt, die in zeitlich aufeinanderfolgenden Iterationen ausgeführt werden.

Phase abhängt, in der die Iteration stattfindet. Diese Strukturierung wird in RUP durch *Work Flows*[1] definiert:

- *Work Flow Business Modeling* und *Work Flow Requirements* entsprechen der Phase Analyse und Definition,
- *Work Flow Analysis and Design* entspricht der Phase Entwurf,
- *Work Flow Implementation* entspricht der Phase Implementierung,
- *Work Flow Test* entspricht der Phase Integration und Test,
- *Work Flow Deployment* entspricht der Phase Einsatz und Wartung.

2.1.1.2 Einordnung der Entwicklungstechniken

Es ist festzustellen, daß die Übergänge zwischen RUP-Phasen managementbezogenen Charakter haben: Es werden Meilensteine definiert, nach deren Erreichen eine jeweils neue RUP-Phase beginnt. Die in Kapitel 1 diskutierten Phasenübergänge haben technologischen Charakter, sie finden innerhalb der Iterationen in einer konkreten RUP-Phase zwischen *Work Flows* statt. Objektorientierte Modellierung ist ein integraler Bestandteil von RUP, diese wird während *Business Modeling Work Flow*, *Requirements Work Flow* sowie *Analysis and Design Work Flow* eingesetzt. Die Entwicklungstechnik automatische Ableitung von Softwarekomponenten kann den Übergang von *Analysis and Design Work Flow* zu *Implementation Work Flow*, *Test Work Flow* und *Deployment Work Flow* in der in Abschnitt 1.6 charakterisierten Art und Weise ermöglichen. Aufgrund der in RUP integrierten Praktik "Modularisierung des zu entwickelnden Softwaresystems" ist der Einsatz von Komponentenarchitekturen als Integrations- und Einsatzumgebung essentiell.

Offensichtlich ist die Nutzung der Entwicklungstechniken objektorientierte Modellierung, automatische Ableitung von Softwarekomponenten und Einsatz von Komponentenarchitekturen im Kontext der Verwendung von RUP möglich und sinnvoll. Insbesondere ist festzustellen, daß RUP extensiv von Iterationen innerhalb der RUP-Phasen Gebrauch macht, und damit die notwendige Entwicklungszeit eines Softwaresystems stark vom zeitlichen Aufwand für *Work-Flow*-Übergänge abhängt.

RUP ist ein allgemeiner Entwicklungsprozeß für Softwaresysteme, er ist weder spezifisch für verteilte Softwaresysteme noch für Telekommunikationssoftwaresysteme. Bei Einsatz der Entwicklungstechnik objektorientierte Modellierung im Kontext von RUP wird es jedoch durch die Definition eines spezifischen Konzeptraumes möglich, die besonderen Anforderungen an die Entwicklung solcher Softwaresysteme zu erfüllen.

2.1.2 Das V-Modell

2.1.2.1 Charakterisierung

Das V-Modell wurde ursprünglich im Auftrag des Bundesministeriums für Verteidigung und in Zusammenarbeit mit dem Bundesamt für Wehrtechnik und Beschaffung in Koblenz von der Industrieanlagen-Betriebsgesellschaft mbH in Ottobrunn bei München erstellt. Im Sommer 1992 wurde es vom Bundesministerium des Innern für den Bereich der Bundesverwaltung übernommen und ist seit Juni 1996 auch dort eine verbindlich einzusetzende Vorschrift. Der Fokus wurde auf Softwaresystementwicklung gelegt, jedoch ist - begründet in der Anwendung im militärischen Kontext - auch die Entwicklung kombinierter Software-/ Hardware-Systeme durch das V-Modell erfaßt, so daß allgemein der Terminus Systementwicklung genutzt wird.

Das V-Modell umfaßt drei Ebenen (vgl. Abb. 9):

1. Neben *Work Flows* zur konstruktiven Entwicklung des Softwaresystems selbst definiert RUP weitere, managementbezogene *Work Flows*. In Bezug auf den Einsatz von Entwicklungstechniken werden hier nur *Work Flows* betrachtet, die der konstruktiven Entwicklung des Softwaresystems dienen.

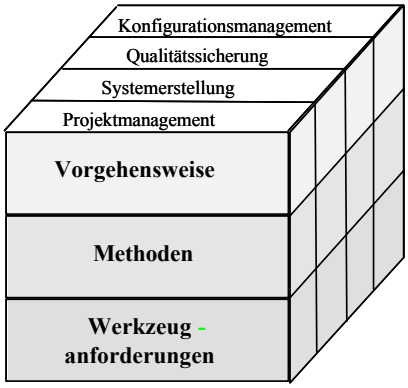


Abb. 9 Struktur des V-Modells

- *Das Prozeßmodell (Vorgehensweise)*

  Diese Ebene legt fest, welche Tätigkeiten im Verlauf der Systementwicklung durchgeführt werden, welche Ergebnisse dabei zu produzieren sind und welche Inhalte diese Ergebnisse haben müssen;

- *Die Zuordnung von Entwicklungstechniken (Methoden)*

  Hier erfolgt eine Bestimmung, mit welchen Methoden die auf der ersten Ebene festgelegten Tätigkeiten durchgeführt werden und welche Darstellungsmittel für die Ergebnisse zu verwenden sind;

- *Die Anforderungen an Entwicklungswerkzeuge (Werkzeuganforderungen)*

  Auf dieser Ebene wird fixiert, welche funktionalen Eigenschaften die Entwicklungswerkzeuge aufweisen müssen, die bei der Systementwicklung eingesetzt werden sollen.

Damit ist umrissen, in welchen Schritten und mit welchen Methoden die Entwicklungsarbeiten auszuführen sind und welche funktionalen Eigenschaften die zum Einsatz kommenden Werkzeuge aufweisen müssen.

Alle Ebenen werden in Tätigkeitsbereiche, die sog. Submodelle gegliedert:

- Projektmanagement,
- Systemerstellung,
- Qualitätssicherung und
- Konfigurationsmanagement.

Im Submodell Systemerstellung sind alle unmittelbar der Erstellung eines Systems dienenden Aktivitäten und die jeweiligen zu erstellenden Entwicklungsdokumente zusammengefaßt. Es enthält die Aktivitäten Anforderungsanalyse, Systementwurf, Software-/Hardware-Anforderungsanalyse, Software-/Hardwareerstellung, Systemintegration und Überleitung in die Nutzung. Diese Aktivitäten haben eine direkte Entsprechung im Phasenmodell, die im folgenden diskutiert wird.

- *Anforderungsanalyse*

  Hier erfolgt die Festlegung der Anforderungen an das zu erstellende System und seine technische und organisatorische Umgebung und die Durchführung einer Bedrohungs- und Risikoanalyse. Die Aktivität Anforderungsanalyse entspricht der Phase Analyse und Definition im Phasenmodell.

- *Systementwurf*

  In dieser Aktivität erfolgt der Entwurf und damit die Spezifikation der Komponenten des Systems. Die Aktivität Systementwurf entspricht der Phase Entwurf im Phasenmodell.

- *Software-/Hardware-Anforderungsanalyse*

  Die technischen Anforderungen an die Software- und gegebenenfalls Hardwarekomponenten werden nochmals präzisiert. Von hier ab spaltet sich der weitere Entwicklungsprozeß in die Software- und Hardwareerstellung. Diese Aktivität kann als einfache Iteration zwischen Entwurfsphase und der Phase Analyse und Definition aufgefaßt werden.

- *Software- und Hardwareerstellung*

  Hier erfolgt die Implementierung der im Systementwurf spezifizierten Software- und Hardwarekomponenten. Diese Aktivität entspricht der Phase Implementierung im Phasenmodell.

- *Systemintegration*

  Während der Systemintegration erfolgt die Integration der verschiedenen Software- und Hardwarekomponenten zum System. Diese Aktivität korrespondiert mit der Integrations- und Testphase des Phasenmodells.

- *Überleitung in die Nutzung*

  In dieser Aktivität werden alle Tätigkeiten beschrieben, die notwendig sind, um ein fertiggestelltes System an der vorgesehenen Einsatzstelle zu installieren und in Betrieb zu nehmen. Die Überleitung des Systems in die Nutzung beinhaltet Aktivitäten, die denen der Phase Einsatz und Wartung entsprechen.

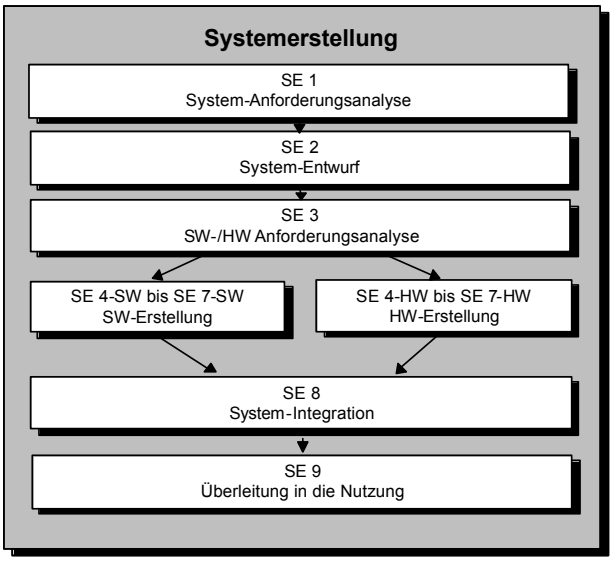


Abb. 10 Das Submodell Systemerstellung des V-Modells

2.1.2.2 Einordnung der Entwicklungstechniken

Das Submodell Systemerstellung ist entsprechend des Phasenmodells ohne Iterationen strukturiert. Eine Besonderheit ist der Übergang zwischen den Aktivitäten Systementwurf und Software-/Hardware-Anforderungsanalyse. Dieser Übergang kann als einfache Iteration zwischen den Phasen Analyse und Definition sowie Entwurf aufgefaßt werden. Ein weiteres Merkmal ist die Berücksichtigung von Hardware im Entwicklungsprozeß.

Offensichtlich ist die Nutzung der objektorientierten Modellierung im Kontext des V-Modells möglich. Der Einsatz der Entwicklungstechnik automatische Ableitung von Softwarekomponenten unterstützt den Über-

gang von der Aktivität Systementwurf zur Aktivität Software- und Hardwareerstellung. Das V-Modell sieht eine Modularisierung des zu entwickelnden Systems in Software- und Hardwarekomponenten vor, dementsprechend sichert der Einsatz einer Komponentenarchitektur eine Reduktion des Aufwandes beim Übergang von der Aktivität Software- und Hardwareerstellung zu den Aktivitäten Systemintegration und Überleitung in die Nutzung - jedenfalls für die entstehenden Softwarekomponenten.

Im Vergleich zu iterativen Entwicklungsprozessen ist der Nutzen des Einsatzes der Entwicklungstechnik automatische Ableitung von Softwarekomponenten bezüglich der Reduktion der Entwicklungszeit jedoch insofern beschränkt, als daß der Übergang von der Phase Entwurf zur Phase Implementierung nur einmal erfolgt. Allerdings ist festzustellen, daß bei einem nicht-iterativen Entwicklungsprozeß der Übergang zwischen den Phasen des Prozesses fehlerfrei erfolgen muß, sich der Einsatz der automatischen Ableitung von Softwarekomponenten also auf die Qualität des Übergangs auswirkt.

2.1.3 Software Process Engineering Metamodel

2.1.3.1 Charakterisierung

Konkrete Softwareentwicklungsprozesse, die in in der Softwareentwicklung tätigen Unternehmen eingesetzt werden, sind oftmals nicht veröffentlichte Firmenrichtlinien. Jedoch hat sich herausgestellt, daß diese Unternehmen auch im Bereich der Softwareentwicklung zusammenarbeiten, demzufolge eine Vereinheitlichung der in Softwareentwicklungsprozessen verwendeten Termini und deren Semantik erforderlich ist. Im November 1999 wurde durch *Object Management Group (OMG)* ein RFP-Dokument (*Request for Proposals*) "*Software Process Engineering (SPE) Management*" [OMG SPE RFP] verabschiedet, in dem nach der Definition der Semantik von Termini gefragt wird, die in den Softwareentwicklungsprozessen der Mitgliedsorganisationen von OMG Verwendung finden. Der dadurch begonnene Standardisierungsprozeß dieser Begriffsbildung hat bereits zu mehreren Antworten geführt, die in einem gemeinsamen Dokument "*The Software Process Engineering Metamodel (SPEM)*" [OMG SPEM 01] der antwortenden Unternehmen zusammengefaßt sind. Dieses Dokument enthält gerade diejenigen Termini mit ihrer Semantik, die in Entwicklungsprozessen der antwortenden Unternehmen Verwendung finden. Mit diesem OMG-Prozeß wird also nicht etwa *ein* Softwareentwicklungsprozeß standardisiert, sondern allgemeine Termini und Mechanismen normiert, die sich in den Prozessen der antwortenden Unternehmen wiederfinden lassen. In [OMG SPEM01] sind die Beziehungen zwischen den zur Normierung vorgeschlagenen Termini und Termini verbreiteter Softwareentwicklungsprozesse zusammengefaßt.

Entscheidend im Kontext dieser Arbeit ist, daß das Konzept Iteration - und somit iterative Softwareentwicklung - in diesem gemeinsamen Dokument enthalten ist, also eine Bedeutung besitzt, die über die Anwendung in für die Allgemeinheit zugänglichen Beschreibungen von Softwareentwicklungsprozessen offenbar hinausgeht.

2.1.3.2 Einordnung der Entwicklungstechniken

Die Einordnung der Entwicklungstechniken in SPEM ist nicht möglich, da SPEM keinen konkreten Entwicklungsprozeß beschreibt - und dies auch nicht das Ziel des RFP-Prozesses ist. Jedoch kann festgestellt werden, daß die aus RUP bekannten Termini und deren Semantik eine Entsprechung in den anderen betrachteten Entwicklungsprozessen haben. Insofern lassen sich die in Abschnitt 2.1.1.2 getroffenen Aussagen auf diese Entwicklungsprozesse und alle weiteren Prozesse übertragen, die sich in das durch SPEM vorgegebene Schema einpassen lassen.

2.2 Entwicklungstechnik Einsatz von Komponentenarchitekturen

Komponentenarchitekturen werden eingesetzt, um die Integration, den Test und die Wartung von Softwarekomponenten verteilter Softwaresysteme zu vereinfachen und die Überführung der entwickelten Softwaresysteme in den Einsatz zu beschleunigen. Die Nutzung einer *standardisierten* Komponentenarchitektur für die Softwarekomponenten verteilter Softwaresysteme leistet darüber hinaus einen wichtigen Beitrag zur Offenheit des Softwaresystems - sie gestattet die Kooperation von Softwarekomponenten *verschiedener* Hersteller, deren Interfaces mittels einer durch die Komponentenarchitektur vorgegebenen Schnittstellenbeschreibungssprache definiert werden. Komponentenarchitekturen erlauben einen vereinheitlichten Zugang zu den Mechanismen der Infrastruktur verteilter Softwaresysteme, so daß in den Realisierungen dieser Softwarekomponenten von konkreten technologischen Aspekten wie Zugang zu persistenten Speichern, Aufbau und Nutzung von Kommunikationskanälen und Registrierung/Auffinden konkreter, während der Abarbeitung von Softwarekomponenten instanziierter Objekte abstrahiert werden kann.

Es ist festzustellen, daß - bereits in der anzustrebenden Offenheit der zu entwickelnden Softwaresysteme begründet - der Einsatz derjenigen Komponentenarchitekturen als Entwicklungstechnik in der betrachteten Domäne zu bevorzugen ist, die zum einen weitgehend standardisiert sind, und zum anderen eine große Bedeutung hinsichtlich ihres derzeitigen Einsatzes erlangt haben. Insofern haben sich in den letzten Jahren drei derartige Komponentenarchitekturen herauskristallisiert: *Common Object Request Broker Architecture* [OMG CORBA], *Enterprise Java Beans* [SunEJB] und *Component Object Model* [MS COM].

2.2.1 Common Object Request Broker Architecture

Im Rahmen der *Object Management Group* wurde die *Common-Object-Request-Broker*-Architektur entwickelt und standardisiert. CORBA beinhaltet die folgenden Bestandteile:

- *Object Request Broker* (ORB) stellen die Basisinfrastruktur für Objekte bereit, die mit anderen Objekten in einer verteilten Umgebung interagieren. Durch ORBs werden diese Interaktionen unabhängig vom Ort der interagierenden Objekte vermittelt (*Location Transparency*). ORBs stellen damit die Grundlage für die Entwicklung von Anwendungen basierend auf verteilten Objekten dar. Darüber hinaus sichern ORBs die Zusammenarbeit der verteilten Bestandteile von Anwendungen in heterogenen und homogenen Umgebungen.
- Durch *Object*-Services wird eine Menge von Diensten (durch Interfacedefinitionen und diese Interfacedefinitionen realisierende Objekte) definiert. Diese Dienste unterstützen Basisfunktionalität zur Nutzung und Realisierung verteilter Anwendungen, und zwar unabhängig von der konkreten Anwendungsdomäne. *Object*-Services sind Bestandteil einer allgemeinen CORBA-Infrastruktur.
- *Common Facilties* beinhaltet eine Menge von Diensten, die von anderen Anwendungen - vergleichbar mit *Object*-Services - genutzt werden können. Im Gegensatz zu *Object*-Services sind sie nicht Bestandteil einer allgemeinen CORBA-Infrastruktur, da sie nicht deren generellen Charakter besitzen - sie sind nicht für alle CORBA-Anwendungen fundamental.
- *Application Objects* sind Bestandteile von Anwendungen, die die CORBA-Infrastruktur nutzen, um miteinander zu interagieren. Diese Bestandteile werden nicht der Standardisierung innerhalb von *Object Management Group* zugeführt.

CORBA definiert ein Objektmodell, das die Semantik eines Objektes, dessen Interface und dessen Realisierung definiert. Zur Definition der Interfaces wird die Beschreibungssprache *Interface Definition Language* definiert, nachfolgend mit CORBA-IDL bezeichnet. CORBA-IDL definiert ein Datentypsystem, das mit dem der Programmiersprachen C++ bzw. JAVA semantisch vergleichbar ist. Interfacedefinitionen in CORBA-IDL beinhalten Interaktionselemente der Arten Operation und Attribut. Für Operationen ist der Mechanismus *Exception* (Ausnahme) definiert. Für CORBA-IDL sind verschiedene Ableitungsregeln in Programmiersprachen (*Language Mappings*) standardisiert. Diese Ableitungsregeln werden auf der Seite der Objektimplementierung benutzt, um für Operationen entsprechendes Verhalten im Kontext von Objekten

zu realisieren. Auf der Seite eines Klienten werden die Ableitungsregeln verwendet, um an Interfacedefinitionen definierte Operationen und Attribute zu nutzen. Eine hervorragende Einführung in CORBA in Kombination mit C++ bietet [HV 99].

CORBA stellt eine Infrastruktur bereit, die sich in idealer Weise als Basis einer Komponentenarchitektur eignet. Diese Infrastruktur bietet Konzepte, die eine offene, portable Lösung zur Unterstützung der operationalen und Signalinteraktion zwischen Komponenten verteilter Softwaresysteme beinhaltet. Die Portabilität von CORBA äußert sich insbesondere in der Normierung der programmiersprachenunabhängigen Definition der Interfaces von Objekten und in den dazugehörigen Ableitungsregeln für eine Vielzahl konkreter Programmiersprachen. Darüber hinaus normiert CORBA den Mechanismus *Portable Object Adapter* (POA), der die Portierung der Realisierung von Objekten und deren Interfaces zwischen unterschiedlichen CORBA-Produkten gewährleistet. CORBA wird bereits in großen Telekommunikationsunternehmen für die Integration von Softwarekomponenten eingesetzt, u.a. in den Unternehmen *Alcatel*, *Deutsche Telekom AG*, *MCI Worldcom* und *NEC* [OOCa].

2.2.2 Enterprise Java Beans

Ihren Ursprung hat die Entwicklung von *Enterprise Java Beans* (*EJB*) in der Architektur des Komponentenmodells für Geschäftsanwendungen "*San Francisco*" von IBM. Kernideen daraus sind in die EJB-Spezifikation eingeflossen, die von *Sun Microsystems* federführend herausgegeben wird. Die z.Z. freigegebene Spezifikation ist die Version 1.1 [SunEJB].

Die Architektur von EJB ist eine Komponentenarchitektur für die Entwicklung und Verteilung von mit JAVA realisierten Softwarekomponenten. EJB geht davon aus, daß ein Entwickler sich nur mit der Realisierung von *Enterprise*-Funktionen in Form eines *Enterprise*-Objekts befaßt. Softwaresysteme, die unter Nutzung von EJB-konformen *Enterprise*-Objekten entwickelt werden, sind skalierbar, im Mehrbenutzerbetrieb sicher einsetzbar und mit einem Transaktionsmechanismus ausgestattet. Diese Mechanismen werden durch die Einsatzumgebung von *Enterprise*-Objekten - als *Container* bezeichnet - bereitgestellt.

Die wichtigsten Eigenschaften von *Enterprise Java Beans* (*EJB*) sind:

- Portabilität der *Enterprise*-Objekte, sichergestellt durch die Konzepte von JAVA selbst und *Container*,
- ausschließliche Konzentration auf die *Enterprise*-Funktionen von *Enterprise*-Objekten vereinfacht die Entwicklung von Softwaresystemen,
- Die Architektur ist unabhängig von der Technologie der benutzten Kommunikationsinfrastrukturen, jedoch nicht von der verwendeten Programmiersprache (JAVA),
- Flexibilität und Skalierbarkeit der resultierenden Softwaresysteme

Die Kommunikation zwischen EJB-Komponenten geschieht mittels JAVA *Remote Method Invocation* (RMI). Um die Zusammenarbeit mit anderen Architekturen und Umgebungen zu ermöglichen, ist eine Abbildung von RMI auf *Internet Inter-ORB Protocol* (*IIOP*) definiert, so daß die Nutzung von CORBA als Kommunikationsinfrastruktur möglich ist und CORBA-basierte Softwarekomponenten in EJB-basierte Softwaresysteme integriert werden können.

EJB stellt - analog zu CORBA - eine Infrastruktur bereit, die die Basis einer Komponentenarchitektur ist, die mit JAVA realisierte Softwarekomponenten zu integrieren gestattet. Diese Infrastruktur stellt eine offene, portable Lösung zur Unterstützung der operationalen Interaktion zwischen Komponenten verteilter Softwaresysteme dar. Im Gegensatz zu CORBA setzt der Einsatz der Komponentenarchitektur EJB die ausschließliche Nutzung von Java als Programmiersprache für die Implementierung voraus. Die grundsätzlichen Konzepte der Komponentenarchitektur EJB werden in der weiteren Normierung von CORBA verallgemeinert [OMG CCM I].

2.2.3 Component Object Model

Component Object Model (*COM*, [Box 99]) wurde durch die Firma *Microsoft Corp.* im Jahre 1993 eingeführt und wird seitdem in den Betriebssystemen dieses Unternehmens als Komponentenarchitektur eingesetzt. COM definiert einen programmiersprachenunabhängigen, objektorientierten, binären Interoperabilitätsansatz als Infrastruktur für die Integration von Softwarekomponenten. COM erlaubt die Nutzung der durch eine Softwarekomponente bereitgestellten Funktionalität unabhängig von der zur Realisierung dieser Komponente verwendeten Programmiersprache.

COM nutzt die Termini Objekt, Klasse, Interface und Komponente. Ein Objekt im Kontext von COM ist eine konkrete Instanz einer namentlich identifizierbaren Klasse, eine Klasse ist hier die Implementierung eines oder mehrerer Interfaces. Jedes Interface repräsentiert in COM eine Gruppierung von in logischer Relation stehenden Funktionen, als Methoden bezeichnet. Ein Interface bezeichnet somit einen streng getypten Kontrakt zwischen dem Nutzer eines Objektes und dem Objekt selbst, das dieses Interface anbietet. COM definiert im Gegensatz zu CORBA kein über Maschinengrenzen hinaus gültiges Referenzkonzept. Anstelle dessen erhält man den Zugang zu Interfaces eines Objektes über das Konzept *Interface Pointer*. Jedes COM-Objekt unterstützt mindestens das Interface **IUnknown**, alle weiteren Interfaces entstehen durch Spezialisierung von **IUnknown**. In COM wird die Schnittstellenbeschreibungssprache *Microsoft Interface Definition Language (MIDL)* genutzt, um konkrete Interfaces mit deren Methoden zu beschreiben. MIDL ist eine objektorientierte Erweiterung der durch *Open Software Foundation* (*OSF*) definierten Sprache *Interface Definition Language*, die im Kontext von *Distributed Computing Environment* (*DCE*) entwickelt wurde. Diese Erweiterungen hatten zum Ziel, Objektorientierung im Kontext verteilter Softwaresysteme zu unterstützen.

Durch die Integration von COM in das weit verbreitete Betriebssystem *Windows* der Firma Microsoft hat sich dieser Ansatz als Komponentenarchitektur zur Integration von Softwarekomponenten auf einer Maschine durchgesetzt. Die Softwarekomponenten von Anwendungen, die auf einer Maschine unter dem Betriebssystem *Windows* ausgeführt werden, interagieren miteinander und mit den Komponenten anderer Anwendungen auf der gleichen Maschine unter Nutzung von COM. Die Definition von *Distributed COM* (DCOM) gilt als Versuch der Firma *Microsoft*, die Nutzung von COM auf die Interaktion zwischen Komponenten auf verschiedenen Maschinen auszudehnen. DCOM konnte sich jedoch bisher nicht als Komponentenarchitektur für verteilte Softwaresysteme - insbesondere im Telekommunikationskontext - durchsetzen, wie durch eine durch das Unternehmen *Sun Microsystems* durchgeführte und veröffentliche Studie mit dem Titel "*Industry Opinions On Enterprise JavaBeansTM (EJBTM) vs. COM+/MTS*" belegt wurde [Sun EJBa]. Insofern betrachten die Autoren die Komponentenarchitekturen COM und DCOM als nicht praktikabel für Integration und Einsatz von Softwarekomponenten verteilter Telekommunikationssoftwaresysteme.

Microsoft hat mit der Ankündigung der als *.net* bezeichneten Architektur die extensive Nutzung von *Internet*-Standards als Grundlage einer neu zu definierenden Komponentenarchitektur erklärt. Eine derartige Strategie würde sicherlich, sofern sie umgesetzt werden kann, eine größere Akzeptanz dieser Komponentenarchitektur für verteilte Softwaresysteme bewirken. Jedoch sind zum gegenwärtigen Zeitpunkt weder die Architektur *.net* noch eine dazugehörige Komponentenarchitektur fertiggestellt.

2.3 Entwicklungstechnik objektorientierte Modellierung

Objektorientierte Modellierung ist seit geraumer Zeit die zentrale Entwicklungstechnik bei der Erfassung von Geschäftsprozessen und dem Entwurf von Softwaresystemen, die diese Geschäftsprozesse automatisieren [Gart99][Gart99a]. In Abschnitt 1.1 wurde die Schlußfolgerung gezogen, daß die Konstruktion eines Konzeptraumes der betrachteten Domäne zentrale Voraussetzung der zielgerichteten Anwendung objektorientierter Modellierung ist. Die Definition eines Konzeptraumes muß dabei sicherstellen, daß solche Konzepte aufgenommen werden, die sich aus den in Abschnitt 1.5 diskutierten Anforderungen an die Entwicklungstechniken ergeben. Im folgenden werden nun Ansätze untersucht, die als Ausgangspunkt der

Definition eines Konzeptraumes für die Entwicklung verteilter Telekommunikationssoftwaresysteme dienen können.

2.3.1 RM-ODP

Eine initiale Definition der zur objektorientierten Modellierung von verteilten, offenen Softwaresystemen einsetzbaren Konzepte findet sich im Referenzmodell für Offene Verteilte Verarbeitung (RM-ODP) [ITUT X.902]. Insbesondere wird der Begriff eines offenen, verteilten Systems präzisiert. Ein solches System (ODP-System) ist danach eine Entität, deren Betrachtung als ein ganzes oder als ein Zusammenschluß von Komponenten von Interesse ist. Die durch das RM-ODP vorgestellte Architektur verfolgt den Ansatz, durch die Definition von fünf Modellklassen jeweils unterschiedliche Systemaspekte zu berücksichtigen. In den einzelnen Modellklassen werden Konzepte zur Modellierung der folgenden Aspekte bereitgestellt:

- *Enterprise*-Modellklasse

  Anwendungsbereich, Zweck und Regeln des Systems;

- *Information*-Modellklasse

  Art und Semantik der durch das System verarbeiteten Informationen;

- *Computational*-Modellklasse

  Funktionale Dekomposition des Systems in eine Menge interagierender Objekte;

- *Engineering*-Modellklasse

  Aspekte der Verteilung von Objekten und der benötigten Infrastruktur zur Unterstützung der Interaktion dieser Objekte;

- *Technology*-Modellklasse

  konkrete Hard- und Software, technologische Aspekte.

Die Konzepte der *Enterprise*-Modellklasse werden in der Phase Analyse und Definition eingesetzt, sie sind also für die Definition eines Konzeptraumes für die objektorientierte Modellierung in der Entwurfsphase von verteilten Telekommunikationssoftwaresystemen nicht relevant. Hingegen sind die Konzepte der *Information*-, *Computational*- und *Engineering*-Modellklasse bezüglich der Konstruktion eines solchen Konzeptraumes zu untersuchen. Konzepte der *Technology*-Modellklasse bilden eine Grundlage von Komponentenarchitekturen, und sind, um die technologische Unabhängigkeit zu sichern, für die Definition eines Konzeptraumes nicht relevant.

Eine umfassende Einführung des RM-ODP findet sich in [Ray 95]. Im Zusammenhang mit der detaillierten Definition des Konzeptraumes für die Modellierung verteilter Telekommunikationssoftwaresysteme werden die Konzepte der Modellklassen des RM-ODP näher untersucht.

2.3.2 TINA Computational Modeling Concepts

Das TINA-Konsortium (*Telecommunications Information Networking Architecture Consortium*, [TINA]) wurde durch Telekommunikationsnetzbetreiber, Softwarehersteller und Hersteller von Telekommunikationsanlagen im Jahre 1993 gegründet und erarbeitete bis zum Jahre 2000 eine Architektur für offene Telekommunikationssoftwaresysteme. Folgende Anforderungen wurden durch das TINA-Konsortium an die zu entwerfende Architektur gestellt [TINA Req]:

- *Interoperabilität*

  Die Architektur soll die Zusammenarbeit von Applikationen bzw. Komponenten von Applikationen über die Grenzen administrativer Bereiche hinaus erlauben.

- *Wiederverwendung von Softwaremodulen, Kompatibilität mit existierenden Telekommunikationssystemen*

  Es soll ermöglicht werden, daß sowohl Spezifikationen als auch Softwarekomponenten wiederverwendet werden können, um sie neue z.B. in neu zu entwickelnde Softwaresysteme zu integrieren. Weiterhin soll es die Architektur ermöglichen, existierende, nicht nach TINA-Prinzipien entworfene Softwarekomponenten in neuen, TINA-konformen Softwaresystemen wiederzuverwenden. TINA-konforme und existierende Telekommunikationssysteme sollen zusammenarbeiten können, die Architektur soll die Migration von existierenden zu TINA-konformen Systemen erlauben.

- *Verteilte Abarbeitung von Applikationen, Unabhängigkeit von Rechentechnik und Kommunikationshardware*

  Die Architektur soll der verteilten Ausführung zugrundeliegende Kommunikationsmechanismen vor den Softwarekomponenten der Systeme verbergen - Abhängigkeiten zwischen Softwaresystemen und den Ressourcen von Maschinen, auf denen Softwarekomponenten ausgeführt werden, sind durch die Architektur zu minimieren.

- *Unterstützung neuer Dienstarten, Managementunterstützung*

  Die Architektur soll komplexe, multimediale Softwaresysteme durch den Einsatz neuer, technologieunabhängiger Dienstmodelle unterstützen. Das Management von Softwaresystemen und verwendeter Hardware soll ermöglicht werden.

- *Dienstqualität*

  Die Architektur soll das Setzen von Attributen für an der Erbringung eines Dienstes beteiligten Objekten behandeln, um bestimmte Anforderungen z.B. an Netzressourcen zu sichern.

Das Kernprinzip für den Entwurf der diesen Anforderungen gerecht werdenden Architektur war die Anwendung der in RM-ODP enthaltenen Modellierungskonzepte. Die Herangehensweise des TINA-Konsortiums war damit der initiale Versuch, diese Modellierungskonzepte bezüglich der Domäne Telekommunikation anzupassen, zu verfeinern bzw. zu erweitern. Insbesondere die für den Entwurf von Softwaresystemen relevanten Konzepte der *Computational*-Modellklasse des RM-ODP wurden genutzt, um die Spezifikation der TINA-C-Architektur zu erstellen. Auf der Basis der *Computational*-Modellierungskonzepte wurde mit *Computational Modeling Concepts* [TINA CMC] ein Konzeptraum für den Entwurf von Telekommunikationssoftwaresystemen geschaffen, für diesen Konzeptraum wurde eine graphische und eine textuelle Notation definiert. Diese Notationen - bezeichnet als ITU-ODL - wurden im Kontext von ITU-T weiterentwickelt und normiert [ITUTZ.130].

Die Definition von Notationen für den durch die *Computational*-Modellklasse implizierten Konzeptraum war im Kontext der hier geführten Diskussion bezüglich der Entwicklungstechnik objektorientierte Modellierung der entscheidende Beitrag, den das TINA-Konsortium in die Weiterentwicklung von RM-ODP eingebracht hat. Das TINA-Konsortium hat allerdings nicht alle, in der *Computational*-Modellklasse von RM-ODP enthaltenen Modellierungskonzepte berücksichtigt. Insbesondere die asynchrone, entkoppelte Interaktion zwischen COs wurde nicht betrachtet, so daß die Notationen ITU-ODL tatsächlich nicht alle, in der *Computational*-Modellklasse enthaltenen Modellierungskonzepte auszudrücken vermag. Es kann festgestellt werden, daß in TINA *Computational Modeling Concepts* eine Teilmenge der durch RM-ODP definierten Modellierungskonzepte der *Computational*-Modellklasse betrachtet wird - es also dementsprechend bei der Definition eines Konzeptraumes für die Entwicklungstechnik objektorientierte Modellierung hinreichend ist, die durch RM-ODP definierte *Computational*-Modellklasse zu untersuchen.

2.3.3 CORBA Component Model

CORBA definiert nicht nur eine Komponentenarchitektur, sondern beinhaltet durch seine semantische Fundierung - das CORBA-Objektmodell - eine Menge von Modellierungskonzepten. Zur Definition eines Konzeptraumes für die objektorientierte Modellierung verteilter Telekommunikationssysteme sind die Konzepte der Interfacebeschreibungssprache CORBA-IDL von Interesse. Das in CORBA-IDL enthaltene Datentyp-

system, das Konzept Interface und Interfacereferenz sowie der Metatyp CORBA-*Object* sind diesbezüglich hervorzuheben. Auf der Grundlage dieser semantischen Fundierung wurde durch OMG eine Erweiterung der Komponentenarchitektur CORBA initiiert [OMG CCM I], die neben dem Objektbegriff auch die Konzepte Softwarekomponente und die Beziehungen zwischen einer Softwarekomponente, den enthaltenen Codemodulen und CORBA-Objekten beinhaltet. Dies CORBA-Erweiterung wurde als CORBA *Components* (CORBA-Komponentenmodell) bezeichnet.

Der Meta-Typ CORBA-*Component* (CORBA-Komponente) definiert einen Kontext für das Anbieten bzw. Benutzen mehrerer CORBA-Interfaces in Kombination mit der Komponentenimplementierung. Grundsätzlich kapselt eine CORBA-Komponente also die Realisierung des Anbietens bzw. Benutzens von CORBA-Interfaces durch Implementierungen und stellt ihren Klienten eine Menge wohldefinierter Verbindungspunkte zur Verfügung. Eine CORBA-Komponente definiert diese Verbindungspunkte durch das *Port*-Konzept. *Port*-Definitionen dienen der Hinterlegung bzw. der Beschaffung von CORBA-Interfacereferenzen. Im CORBA-Komponentenmodell werden *Port*-Definitionen entsprechend den Interaktionsarten operationale Interaktion und Signalinteraktion spezialisiert. Dementsprechend enthält das CORBA-Komponentenmodell:

- Die *Port*-Spezialisierung *facet* für Referenzen namentlich unterscheidbarer operationaler CORBA-Interfaces, die von einer CORBA-Komponente gegenüber ihren Klienten angeboten werden,
- Die *Port*-Spezialisierung *receptacle* als Beschreibung der Möglichkeit, Referenzen namentlich unterscheidbarer CORBA-Interfaces an einer CORBA-Komponente zu hinterlegen, die dann durch eine CORBA-Komponente genutzt werden können,
- Die *Port*-Spezialisierung *event source* als Beschreibung der Fähigkeit einer CORBA-Komponente, Signale eines spezifizierten Typs via namentlich unterscheidbaren Verbindungspunkten an interessierte Konsumenten versenden zu können,
- Die *Port*-Spezialisierung *event sink* als Beschreibung eines namentlich unterscheidbaren Verbindungspunktes, an dem eine CORBA-Komponente Signale eines spezifizierten Typs empfangen kann.

Das *Port*-Konzept widerspiegelt die Fähigkeit einer CORBA-Komponente, im Gegensatz zum Meta-Typ CORBA-*Object*, mehr als ein CORBA-Interface anbieten bzw. nutzen zu können. Weitere, mit dem CORBA-Komponentenmodell eingeführte Konzepte beinhalten die für Klienten nutzbare Identifizierbarkeit von CORBA-Komponenten (*primary key*) sowie den neuen Meta-Typ *home*, dessen Instanzen Operationen zur Erzeugung und zum Auffinden (*home finder*) von CORBA-Komponenten enthalten.

Die Entwicklung des CORBA-Komponentenmodells wurde sehr stark von EJB beeinflußt. Es kann festgestellt werden, daß das CORBA-Komponentenmodell die Komponentenarchitektur EJB in folgenden Punkten erweitert:

- Die Implementierungssprache für CORBA-Komponenten ist nicht auf Java beschränkt, vielmehr kann jede Programmiersprache genutzt werden, für die CORBA-IDL-Ableitungsregeln definiert sind,
- CORBA-Komponenten können - im Gegensatz zu EJB - mehr als ein Interface anbieten bzw. nutzen,
- CORBA-Komponenten können Signalinteraktion unterstützen.

Das CORBA-Komponentenmodell beinhaltet weiterhin Konzepte, die nicht nur die in Softwarekomponenten enthaltenen Codemodule, sondern auch Softwarekomponenten selbst beschreiben. Durch die Definition des CORBA-Komponentenmodells ist sichergestellt, daß diese Softwarekomponenten Interfaces bereitstellen, die die Implementierung und Integration von Softwarekomponenten sowie deren Einsatz und Wartung durch generische, nicht komponentenspezifische Entwicklungswerkzeuge erlauben. Diese Konzepte sind nicht telekommunikationsspezifisch, jedoch stellen sie einen initialen Ansatz zur technologischen Vereinheitlichung der Sicht auf Softwarekomponenten und deren Interfaces dar, und dienen somit als wichtige Basis der Definition eines Konzeptraumes für die Entwicklung verteilter Telekommunikationssoftwaresysteme. Zum gegenwärtigen Zeitpunkt ist allerdings keine praktische Realisierung des CORBA-Komponentenmodells verfügbar, so daß der Nachweis der Wirkungsfähigkeit der Konzepte nicht erbracht werden

konnte. Aufgrund der konzeptionellen Nähe von EJB und CORBA-Komponenten und der Erweiterungen, die das CORBA-Komponentenmodell gegenüber EJB definiert (das CORBA-Komponentenmodell ist konzeptionell eine echte Obermenge von EJB) ist es hinreichend, bei der Definition eines Konzeptraumes das CORBA-Komponentenmodell zu untersuchen.

2.4 Fazit

Die Diskussion der Entwicklungsprozesse zeigt, daß die Integration der Entwicklungstechniken objektorientierte Modellierung, automatische Ableitung von Softwarekomponenten und Einsatz von Komponentenarchitekturen in diese Prozesse möglich und sinnvoll ist. Die Definition eines domänenspezifischen Konzeptraumes für die objektorientierte Modellierung von Softwaresystemen erhöht die erreichbare Qualität und Präzision der Spezifikation dieser Systeme und ermöglicht zudem die automatische Ableitung von Softwarekomponenten. Insbesondere bei iterativen Prozessen ist die Minimierung des zeitlichen Entwicklungsaufwandes durch den integrativen Einsatz der Entwicklungstechniken objektorientierte Modellierung, Einsatz von Komponentenarchitekturen und automatische Ableitung von Softwarekomponenten hervorzuheben. Der Einsatz von Komponentenarchitekturen erlaubt die flexible Integration der Softwarekomponenten zu Softwaresystemen und die rasche Übernahme dieser Systeme in den Einsatz. Die Voraussetzung für diese Flexibilität ist dann gegeben, wenn die Spezifikation des Softwaresystems nicht nur zur Automatisierung des Übergangs in die Implementierungsphase genutzt wird, sondern auch der Automatisierung des Übergangs in die Integrations- und Einsatzphase dient (vgl. Phasenübergänge in Kapitel1).

De jure bzw. *de facto* standardisierte Komponentenarchitekturen, die derzeit eine praktische Bedeutung erlangt haben, wurden bezüglich ihrer Eignung als Integrations- und Einsatzumgebung analysiert. Es kann festgestellt werden, daß CORBA und dessen Erweiterung CORBA *Components* ein vielversprechender Ansatz ist. Allerdings sind auch hier wichtige Anforderungen aus der Telekommunikationsdomäne noch nicht erfüllt, wie z.B. die fehlende Unterstützung der *Continuous-Media*-Interaktionsart und die Unterstützung von *Gütebeschreibung und Garantie*.

Um die Entwicklungstechniken objektorientierte Modellierung und automatische Ableitung von Softwarekomponenten mit dem Einsatz von Komponentenarchitekturen zu integrieren, ist die Definition eines Konzeptraumes für die Entwicklung verteilter Telekommunikationssysteme fundamental. Obwohl der Einsatz von CORBA als Komponentenarchitektur derzeit zu favorisieren ist, können zukünftig auch in der Telekommunikation alternative Komponentenarchitekturen zum Einsatz kommen (z.B. *.net*). Insofern ist anzustreben, daß die Definition eines Konzeptraumes *unabhängig* von einer verwendeten Komponentenarchitektur vorgenommen wird.

Des weiteren ist es essentiell, daß alle beinhalteten Konzepte durch die Entwicklungstechnik automatische Ableitung von Softwarekomponenten berücksichtigt werden. Charakteristika der eingesetzten Komponentenarchitekturen werden dann nur durch diese Entwicklungstechnik behandelt. Der Vorteil dieser Herangehensweise ist offensichtlich: Objektorientierte Modelle, die in der Spezifikation eines Softwaresystems zusammengefaßt sind, können wiederverwendet werden, auch wenn sich die in der Einsatzumgebung verwendete Komponentenarchitektur mit der Zeit verändert.

Das Prinzip der Trennung von konzeptioneller Fundierung (Konzeptraum) der objektorientierten Modellierung von Softwaresystemen einerseits und der verwendeten Komponentenarchitektur, die zur Integration der Softwarekomponenten des Systems eingesetzt wird, andererseits sowie deren Zusammenführung durch die automatische Ableitung von Softwarekomponenten ist auch die Basis der im Entstehen begriffenen *Model Driven Architecture* (*MDA,* [OMG MDA]) der *Object Management Group*. MDA wird als neue Basisarchitektur von OMG entworfen. Das Ziel dabei ist die Integration von OMG-Standards zur objektorientierten Modellierung (z.B. UML) mit Komponentenarchitekturen (z.B. CORBA) zur Anwendung in verschiedensten Domänen, z.B. Telekommunikation.

Derzeit existiert dieser integrative Gedanke jedoch nur als Idee. Diese Arbeit ist die erste konkrete Realisierung der *Model Driven Architecture* für die Entwicklung von verteilten Telekommunikationssoftwaresystemen. Der dazu notwendige Konzeptraum entsteht durch Konstruktion auf der Grundlage der in Abschnitt 2.3 diskutierten Ansätze. Des weiteren werden eine auf CORBA und CORBA *Components* basierende Komponentenarchitektur konzipiert und Ableitungsregeln formuliert, die alle Konzepte des Konzeptraumes auf Mechanismen dieser Komponentenarchitektur abbilden.

Die in dieser Arbeit vorgestellten Entwicklungstechniken werden als *CORE* (*Components Rapid Engineering*) bezeichnet:

- Der Konzeptraum wird $CORE_{\text{CEPT}}$ genannt,
- die Notationen werden unter der Bezeichnung $CORE_{\text{TATIONS}}$ zusammengefaßt,
- die Ableitungsregeln werden als $CORE_{\text{MAP}}$ bezeichnet und
- die Komponentenarchitektur erhält den Namen $CORE_{\text{WARE}}$.

KAPITEL 3

Konzeption der Entwicklungstechniken

Die Betrachtungen der Ausgangslage und die daraus gezogenen Schlußfolgerungen haben die besondere Bedeutung von $CoRE_{CEPT}$ als Grundlage der Entwicklungstechniken für die Entwicklung von verteilten Telekommunikationssoftwaresystemen gezeigt. In diesem Kapitel wird $CoRE_{CEPT}$ konstruiert und die auf dieser konzeptionellen Basis vollzogene Integration der anderen Bestandteile von *CoRE* erläutert. Eine Formalisierung von $CoRE_{CEPT}$ erfolgt in Kapitel3 von [CoRE II], bisher definierte Notationen von $CoRE_{TATIONS}$ werden in Kapitel5 von [CoRE II] vorgestellt. $CoRE_{MAP}$ sowie die CORBA-basierte Komponentenarchitektur $CoRE_{WARE}$ sind in Kapitel1 und Kapitel2 von [CoRE III] detailliert beschrieben.

3.1 Konstruktion von $CoRE_{CEPT}$

Ausgangspunkt für die Definition des Konzeptraumes sind die im Referenzmodell für ODP definierten Konzepte der *Computational*-Modellklasse sowie deren Spezialisierung für die Telekommunikationsdomäne in TINA *Computational Modeling Concepts* [TINA CMC] (vgl. Abschnitt 2.3.1 und Abschnitt 2.3.2). Konkret stehen dabei die der Beschreibungssprache ITU-ODL (ITU *Object Definition Language* [ITUT Z.130]) zugrundeliegenden Konzepte als Umsetzung und Weiterentwicklung von TINA *Computational Modeling Concepts* im Mittelpunkt der Betrachtungen. Da die ITU-ODL-Konzepte sich ausschließlich auf strukturelle Aspekte von CO-Typen und deren Interfaces beschränken, werden in $CoRE_{CEPT}$ Erweiterungen bezüglich der Realisierung von CO-Typen durch Artefakte und deren Zusammenfassung in Softwarekomponenten vorgenommen. Die Konstruktion des Konzeptraumes wird so gestaltet, daß die Integration weiterer, anwendungsspezifischer Konzepte ermöglicht wird - die Mechanismen, die zur Sicherung dieser fundamentalen Eigenschaft von $CoRE_{CEPT}$ eingesetzt wurden, werden in Kapitel 2 von [CoRE II] diskutiert. Dies ist insbesondere für die Sicherung der Anforderung nach konzeptioneller Offenheit von *CoRE* notwendig.

3.1.1 Objektorientierte Konstruktion von $CoRE_{CEPT}$

$CoRE_{CEPT}$ definiert einen Konzeptraum für die objektorientierte Modellierung verteilter Telekommunikationssoftwaresysteme, d.h. auf dieser Basis können objektorientierte Modelle solcher Systeme konstruiert werden. Es ist insofern konsequent, daß zur Konstruktion von $CoRE_{CEPT}$ selbst objektorientierte Modellierung eingesetzt wird. Dazu werden elementare Konzepte verwendet, die die objektorientierten Prinzipien Klassifizierung, Instanziierung, Generalisierung, Spezialisierung, Komposition, Dekomposition und Assoziation beschreiben.

- *Klasse/Objekt*

  Eine Klasse ist die Beschreibung einer Menge von Objekten mit denselben Merkmalen. Klasse ist das elementare Konstruktionskonzept, es setzt das Prinzip Klassifizierung um. Ein Objekt ist Instanz einer Klasse.

- *Generalisierungsrelation*

  Eine Generalisierungsrelation ist eine unidirektionale Relation zwischen zwei Klassen. Sie drückt die Spezialisierung einer Klasse durch die andere Klasse aus, wobei die spezialisierte Klasse die Eigenschaften der allgemeineren Klasse erbt. Generalisierungsrelation setzt die Prinzipien Generalisierung und Spezialisierung um.

- *Assoziation*

  Eine Assoziation ist eine Strukturbeziehung zwischen zwei Klassen, wobei im Falle der Instanziierung dieser Klassen von der Instanz der einen Klasse zur Instanz der anderen Klasse navigiert werden kann. Assoziation setzt das Prinzip Assoziation um.

- *Aggregation*

  Im Gegensatz zur Assoziation bedeutet die Aggregation eine strengere, unidirektionale, strukturelle Beziehung zwischen Klassen im Sinne einer Teil-Ganzes-Beziehung. Aggregation setzt die Prinzipien Komposition und Dekomposition um.

Abb. 11 illustriert das für die Konstruktion von $CoRE_{CEPT}$ angewendete Prinzip: Konzepte, die in $CoRE_{CEPT}$ aufgenommen werden, sind Instanzen allgemeiner objektorientierter Konzepte. Auf der Grundlage von $CoRE_{CEPT}$ Konzepten werden wiederum durch Instanziierung Elemente eines objektorientierten Modells des zu entwickelnden verteilten Telekommunikationssoftwaresystems gewonnen.

Die nachfolgende Präsentation des Konzeptraumes folgt der Historie seiner Konstruktion und beginnt mit den aus ITU-ODL übernommenen, in der Praxis bewährten Konzepten zur Modellierung von CO-Typen und der operationalen Interaktionen ihrer Instanzen (Abschnitt 3.1.2 bis Abschnitt 3.1.6). Die in Abschnitt 2.2 vorgestellten Komponentenarchitekturen beruhen in ihrer technologischen Umsetzung auf diesen elementaren Konzepten. Anschließend werden über ITU-ODL hinausgehende Konfigurationsaspekte eingeführt, die bereits initial im CORBA-Komponentenmodell enthalten sind und zur Erfüllung der Anforderung nach Offenheit Bestandteil des Konzeptraumes sind (Abschnitt 3.1.8). Um die telekommunikationsspezifischen Anforderungen zur Unterstützung von *Continuous-Media*- und Signalinteraktionsarten zu erfüllen, wird das Interfacekonzept aus ITU-ODL so verallgemeinert, daß alle vorgestellten Interaktionsarten unterstützt werden (Abschnitt 3.1.10 bis Abschnitt 3.1.13). $CoRE_{CEPT}$ erfaßt zusätzlich Aspekte der Implementierung des Verhaltens von CO-Typen durch die Aufnahme von Artefakten. Es wird gezeigt, daß sich dadurch gerade die Anforderungen an flexible Adaptierbarkeit, Skalierbarkeit und Offenheit von Softwaresystemen und ihren Komponenten durch geeignete Modelle realisieren lassen (Abschnitt 3.1.15 bis Abschnitt 3.1.17). Abschließend werden Konzepte eingeführt, die zur Unterstützung des automatisierten *Deployment*-Vorganges (Abschnitt 3.1.19 und Abschnitt 3.1.20) sowie zur Modellierung nicht-funktionaler Anforderungen in $CoRE_{CEPT}$ aufgenommen wurden. Die Integration dieser Konzepte dient der Berücksichtigung der Anforderungen nach Gütebeschreibung und Garantie sowie zur flexiblen Integration von Systemkomponenten (Abschnitt 3.1.22 und Abschnitt 3.1.23).

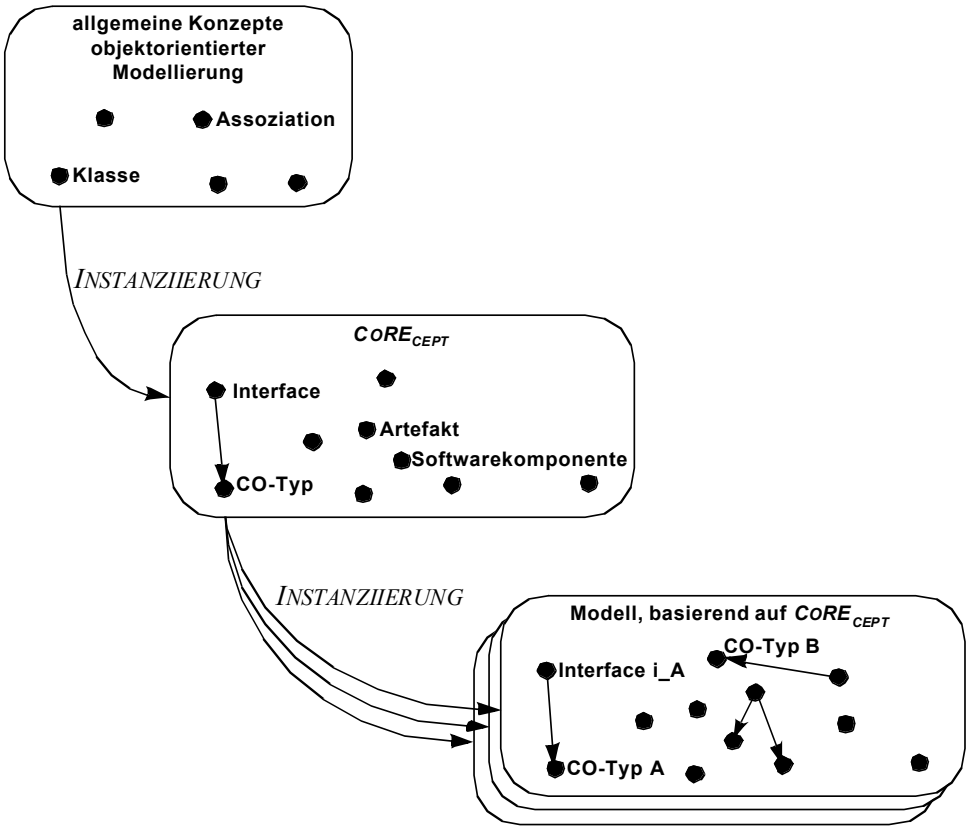


Abb. 11 Objektorientierte Konstruktion von CoRE$_{CEPT}$

3.1.2 Datentyp

CHARAKTERISIERUNG. Datentyp ist Instanz von Klasse. Dieses Konzept wird zur strukturierten Repräsentation von Informationen benutzt, die durch ein Softwaresystem manipuliert werden können. In *CoRE$_{CEPT}$* wird Datentyp durch eine Menge von Datentypen (Datentypsystem) unter Anwendung der Generalisierungsbeziehung spezialisiert. Das derzeit in *CoRE$_{CEPT}$* enthaltene Datentypsystem ist CORBA-IDL.

MOTIVATION. Die Präzisierung von Datentypen, die in Entwurfsmodellen Verwendung finden können, ist essentiell für die Definition aller weiteren Konzepte. Datentypen bilden die Grundlage für die Erstellung von Informationsmodellen. Es existieren eine Reihe von Datentypmodellen als internationale Standards bzw. Teile internationaler Standards, die sich im Kontext dieser Arbeit generell einsetzen lassen: *Abstract Syntax Notation One* (ASN.1) [ITUT X.608], ITU *Object Definition Language* (ITU-ODL) [ITUT Z.130], CORBA *Interface Definition Language* (CORBA-IDL) [OMG CORBA], HTTP *Next Generation* (HTTP-NG) [W3C HTTPNG] und *Extensible Markup Language Data Types* [W3C XML DT]). Exemplarisch findet im Rahmen von *CoRE$_{CEPT}$* das Datentypmodell von CORBA-IDL [OMG CORBA] als Basis von ITU-ODL Anwendung. Dieses ist trotz seiner konzeptionellen Nähe zu weit verbreiteten Programmiersprachen (z.B. C++, JAVA) in dem Sinne programmiersprachenunabhängig, daß Ableitungsregeln in eine Vielzahl von Programmiersprachen definiert und normiert sind. Darüber hinaus ist der Einsatz von CORBA-IDL auch in CORBA-fremden Umgebungen unterstützt (XML [W3CXML], COM/DCOM [MS COM]). Der Austausch von Instanzen von CORBA-IDL-Datentypen (d.h. konkreter Daten) innerhalb eines verteilten Softwaresystems ist durch die Definition der Serialisierungsvorschrift *Common Data Representation* (CDR)

unterstützt und als Teil des CORBA-IDL-Standards [OMGCORBAIDL] normiert. CORBA-IDL wird im Rahmen der Domäne Telekommunikation weit verbreitet eingesetzt (z.B. in der Definition der Standards des TINA-Konsortiums [TINASA]). Die Verwendung des Datentypmodells von CORBA-IDL gestattet die Nutzung von Grunddatentypen wie natürliche Zahlen, reelle Zahlen und Zeichenketten, strukturierten Datentypen wie Strukturen und Sequenzen sowie Referenzen.

HERKUNFTSQUELLE. Das Konzept Datentyp in $CoRE_{CEPT}$ wurde aus ITU-ODL übernommen und basiert auf CORBA-IDL.

3.1.3 Namensraum

CHARAKTERISIERUNG. Namensraum ist Instanz von Klasse, Instanzen von Namensraum werden genutzt, um die Eindeutigkeit von benannten Elementen in einem Entwurfsmodell zu sichern. Dieses Konzept besitzt die gleiche Semantik wie Namensäume in gängigen Programmiersprachen wie z.B. C++.

MOTIVATION. Alle konkreten Instanzen der Konzepte des Konzeptraumes innerhalb von Entwurfsmodellen müssen eindeutig identifizierbar sein. Diese Eigenschaft wird sowohl für die Präsentation der Konzeptinstanzen mittels einer geeigneten Notation als auch für die Auswertung und Informationsbeschaffung über das Entwurfsmodell im Kontext der automatischen Ableitung von Softwarekomponenten benötigt. Die Identifikation erfolgt anhand eines Namens, den jede Instanz eines Konzeptes in einem Entwurfsmodell besitzt.

HERKUNFTSQUELLE. Das Konzept Namensraum in $CoRE_{CEPT}$ wurde aus ITU-ODL übernommen und basiert auf RM-ODP.

3.1.4 Operation, Ausnahme und Parameter

CHARAKTERISIERUNG. Operation, Ausnahme und Parameter sind Instanzen von Klasse. Die Definition von konkreten Operationen und Parametern als Instanzen der Konzepte Operation und Parameter in Entwurfsmodellen beschreibt operationalen Informationsaustausch. Eine Instanz von Operation enthält eine Menge von Instanzen von Parameter sowie eine Menge von Terminierungen. Eine ausgezeichnete Terminierung kann durch Angabe einer Instanz von Datentyp als Rückgabetyp beschrieben werden. Für den Fall von Fehlern während der Ausführung von Operationen können Instanzen von Ausnahme als Terminierungen definiert werden.

MOTIVATION. Während durch die Definition eines Datentypsystems die Beschreibung von Informationselementen und ihren Relationen in einem Entwurfsmodell ermöglicht wird, gestattet die Instanziierung von Operation die Beschreibung der Informationsmanipulation durch operationale Interaktion.

HERKUNFTSQUELLE. Die Konzepte Operation, Ausnahme und Parameter in $CoRE_{CEPT}$ wurde aus ITU-ODL übernommen und basiert auf RM-ODP.

3.1.5 Interfacetyp

CHARAKTERISIERUNG. Interfacetyp ist Instanz von Klasse. Instanzen von Interfacetyp aggregieren eine Teilmenge potentieller Interaktionen im Kontext von CO-Typen. Interfaces als Instanzen der konkreten Instanzen von Interfacetyp sind referenzierbare Interaktionskontexte eines COs.

MOTIVATION. Während mit Instanzen von Operation (d.h. konkreten Operationen im Entwurfsmodell) potentieller Informationsaustausch beschrieben werden kann, gestattet die Definition von Instanzen von Interfacetyp (d.h. konkrete Interfacetypen im Entwurfsmodell) gerade die Zusammenfassung von Operationen zu benannten, identifizierbaren Endpunkten von Interaktion im Sinne von [ITUT X.902]. Interfacetypen stellen dabei einen gemeinsamen Kontext aller von ihnen aggregierten Operationen her (Interaktionskontext). Das Konzept Interfacetyp ist fundamental für die objektorientierte Modellierung offener, verteilter Softwaresysteme, da es die funktionale Dekomposition des Softwaresystems in autonome, interagierende Bestandteile zu beschreiben gestattet. Ausschließlich durch Verwendung des Konzeptes Interfacetyp lassen sich offene Systeme beschreiben (vgl. [Szy 99]). Es sei darauf hingewiesen, daß das Konzept Interfacetyp im weiteren um nicht-operationale Interaktionsarten erweitert wird.

HERKUNFTSQUELLE. Das Konzept Interfacetyp in *CoRE$_{CEPT}$* wurde aus ITU-ODL übernommen und basiert auf RM-ODP.

3.1.6 CO-Typ, Supports- und Requires-Relation

CHARAKTERISIERUNG. CO-Typ ist Instanz von Klasse, *Supports*- und *Requires*-Relation sind Instanzen von Assoziation zwischen CO-Typ und Interfacetyp. Für CO-Typen kann in einem Entwurfsmodell eine Menge von Interfacetypen spezifiziert werden, die durch konkrete COs für deren Umgebung zur Nutzung bereitgestellt werden (Instanz von *Supports*-Relation zwischen CO-Typ und bereitgestelltem Interfacetyp). Darüber hinaus kann eine weitere Menge von Interfacetypen spezifiziert werden, die Instanzen des CO-Typs von ihrer Umgebung erwarten (Instanz von *Requires*-Relation zwischen CO-Typ und erwartetem Interfacetyp). Zur Ermöglichung des Zugriffs auf COs existiert eine Generalisierungsrelation zwischen CO-Typ und Interfacetyp.

MOTIVATION. Bei der funktionalen Dekomposition eines verteilten Softwaresystems werden dessen autonome, über Interfaces interagierende Bestandteile (CO-Typen) unter Verwendung des Konzeptes CO-Typ modelliert. Ein CO ist Instanz eines CO-Typs und unterscheidbar von allen anderen COs (Identität), sowie charakterisiert durch sein Verhalten und seinen Zustand. CO-Typen dienen zur Instanziierung von funktionalen Entitäten eines verteilten Softwaresystems, die letztlich dessen Systemzweck erfüllen.

HERKUNFTSQUELLE. Das Konzept CO-Typ in *CoRE$_{CEPT}$* wurde aus ITU-ODL übernommen und basiert auf RM-ODP.

3.1.7 Diskussion

Mit den bisher eingeführten Konzepten lassen sich gerade diejenigen Entwurfsmodelle definieren, die sich auch mittels der konkreten Syntax von ITU-ODL darstellen lassen. Mit diesen Konzepten läßt sich formulieren, daß ein CO ein Interface für seine Umgebung bereitstellt bzw. von seiner Umgebung benötigt. Jedoch reichen diese Konzepte nicht aus, um zu modellieren, auf welche Weise die Bereitstellung bzw. Nutzung von Interfaces durch CO-Typen erfolgt. Zur Erfassung dieser Aspekte wurde das durch das CORBA-Komponentenmodell eingeführte *Port*-Konzept adaptiert und in einer erweiterten Form in *CoRE$_{CEPT}$* integriert.

3.1.8 Port-Definition, Provided- und Used-Port-Definition

CHARAKTERISIERUNG. *Port*-Definition ist Instanz von Klasse, *Provided*- und *Used-Port*-Definition sind Spezialisierungen von *Port*-Definition. Diese Definitionen dienen zur Beschreibung von Konfigurationsaspekten für CO-Typen. Konkrete (namentlich identifizierbare) *Port*-Definitionen im Kontext eines CO-Typs werden zur Modellierung der Hinterlegbarkeit bzw. Beschaffbarkeit von Referenzen auf Interfaces (Interfacerefe-

renzen) eingesetzt. *Port*-Definitionen basieren entweder auf einer *Supports*-Relation (Beschaffung von Interfacereferenzen durch die Umgebung eines COs) bzw. auf einer *Requires*-Relation (Hinterlegung von Interfacereferenzen durch die Umgebung eines COs). Auf einer *Supports*-Relation basierende *Port*-Definitionen werden als *provided port*, auf einer *Requires*-Relation basierende *Port*-Definitionen als *used port* bezeichnet. Konkrete *Port*-Definitionen in Entwurfsmodellen können die Auszeichnung *multiple* erhalten. An COs, für deren CO-Typ *Port*-Definitionen mit der Auszeichnung *multiple* vorgenommen wurden, können eine Vielzahl von Interfacereferenzen unter dem gleichen *Port*-Namen hinterlegt bzw. beschafft werden. Diese *Port*-Definitionen werden hier auch als *Multiple-Port*-Definitionen bezeichnet. *Port*-Definitionen, die nicht die Auszeichnung *multiple* tragen, werden analog *Single-Port*-Definitionen genannt.

MOTIVATION. Die Konzepte *Provided-* und *Used-Port*-Definition sind semantisch vergleichbar mit den Konzepten *facet* und *receptacle* des CORBA-Komponentenmodells [OMG CCMI]. Während [OMG CCM I] auf statische Konfigurationen von mit dem Konzept CO-Typ vergleichbaren CORBA-Komponenten beschränkt ist, unterstützt *CORE* darüber hinaus dynamische Konfigurationsaspekte.

Durch die Angabe von *Port*-Definitionen im Entwurfsmodell eines verteilten Softwaresystems wird das Konfigurationsmanagement der durch CO-Typen modellierten autonomen Systembestandteile wesentlich erleichtert. In verteilten Softwaresystemen, die auf offenen Komponentenarchitekturen wie CORBA basieren, ist gerade das Konfigurationsmanagement essentiell, aber durch keine normierte kommerzielle Lösung ausreichend, d.h. entsprechend obiger Semantik erfaßt. Mit dem hier beschriebenen Ansatz läßt sich anwendungsunabhängiges (generisches) Konfigurationsmanagement von verteilten Softwaresystemen realisieren. Konfigurationsmanagementwerkzeuge können auf der Basis von Entwurfsmodellinformationen über *Port*-Definitionen Interfacereferenzen zwischen COs zur Ausführungszeit austauschen (vgl. Abb. 12). Eine wichtige Voraussetzung dafür ist die Verfügbarkeit von im Entwurfsmodell hinterlegten Informationen über CO-Typen, *Port*-Definitionen etc. während der Einsatzphase.

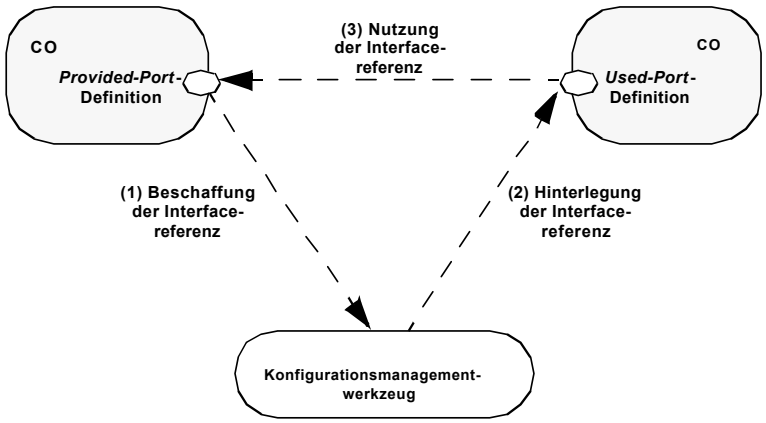


Abb. 12 Konfigurationsmanagement

HERKUNFTSQUELLE. Die Konzepte *Port*-Definition, *Provided-* und *Used-Port*-Definition in $CORE_{CEPT}$ basieren auf den Konzepten *receptacle* und *facet* aus dem CORBA-Komponentenmodell und werden durch *CORE* verallgemeinert (vgl. Abschnitt 2.3.3).

3.1.9 Diskussion

Die bisher eingeführten Konzepte erlauben die Modellierung der autonomen Bestandteile, in die ein verteiltes Softwaresystem während der funktionalen Dekomposition zerfällt, sowie die Modellierung der potentiellen Interaktionen zwischen diesen Bestandteilen.

Allerdings können mit diesen Konzepten nur operationale Interaktionen zwischen COs in Entwurfsmodellen ausgedrückt werden. Entsprechend den in Abschnitt 1.5 formulierten Anforderungen an die Entwicklungstechniken ist aber die Unterstützung der Modellierung von Signalinteraktion und *Continuous-Media*-Interaktion essentiell. In bisherigen Ansätzen ([ITUT X.903], [OMG CCM I], [TINA CMC]) zur Modellierung verteilter Softwaresysteme werden unterschiedliche Interaktionsarten durch verschiedene Arten von Interfacetypen erfaßt. Für operationale Interaktion wird in diesen Ansätzen das Konzept „operationaler Interfacetyp", für Signalinteraktion das Konzept „Signalinterfacetyp" und für *Continuous-Media*-Interaktion das Konzept „*Stream*-Interfacetyp"definiert.

Diese Ansätze haben sich bezüglich dieser konzeptionellen Trennung als nicht praktikabel erwiesen. Den Hauptgrund dafür sehen die Autoren darin, daß das Konzept des Interfacestyps als Interaktionskontext durch diese Trennung zerstört wird. Konkrete Anwendungen, und damit deren Modellierung erfordern aber in der Regel die Homogenität des Prinzips Interaktionskontext *trotz* unterschiedlicher Interaktionsarten. Unterschiedliche Interaktionsarten werden in Softwarekomponenten technologisch unterschiedlich behandelt (vgl. Abschnitt 2.2), dies darf aber keine Implikationen auf die Modellierung bewirken. Die Autoren definieren die Semantik des Konzeptes *Port* als namentlich identifizierbaren Zugangspunkt zu einer konkreten Teilmenge der Funktionalität von CO-Typen. Diese Teilmenge ist durch Interfacetypen wohldefiniert.

CoRE unterstützt die Modellierung unterschiedlicher Interaktionsarten im Kontext eines Interfacetyps. Die Technologieimplikationen werden vollständig durch die automatische Ableitung von Softwarekomponenten aus Entwurfsmodellen umgesetzt. Analog zur Modellierung operationaler Interaktion (Operation) werden zunächst die atomaren Konzepte zur Modellierung der Signal- und *Continuous-Media*-Interaktion eingeführt, bevor die Definition des Konzeptes Interfacetyp entsprechend erweitert werden kann. Für die Signalinteraktion wurden dazu die initialen Ansätze des CORBA-Komponentenmodells (vgl. Abschnitt 2.3.3) erweitert, für *Continuous-Media*-Interaktion hingegen wurden die an der Humboldt-Universität zu Berlin, Institut für Informatik, Lehrstuhl für Systemanalyse, Modellierung und Simualtion entwickelten Konzepte von Medien-*Delivery*-Plattformen [TTK+ 00][TTK+00a][KT 99] integriert und ausgebaut.

3.1.10 Signaltyp und Signalparameter

CHARAKTERISIERUNG. Signaltyp ist Instanz von Klasse. Signalparameter spezialisiert den CORBA-IDL-Datentyp *Value*-Typ, der in *CoRE$_{CEPT}$* enthalten ist, da das Datentypsystem von CORBA-IDL in *CoRE$_{CEPT}$* übernommen wurde. Konkrete Signaltypen als Instanzen dieses Konzeptes beschreiben eine Menge von Signalen, die zwischen COs ausgetauscht werden. Signale als Instanzen von Signaltypen transportieren aktuelle Parameter als Instanzen von Signalparameter.

MOTIVATION. Das Konzept Signaltyp bildet die Grundlage für die Modellierung von Signalinteraktionen, d.h. asynchronem, entkoppelten Austausch atomarer Nachrichten. Im Gegensatz zu [OMG CCM I] wird hier bewußt zwischen Signaltypen und den durch Signale transportierten Instanzen von Signalparameter getrennt. Unterschiedliche Signaltypen können unterschiedlichen Ausführungszeitbeschränkungen unterliegen, die im Entwurfsmodell erfaßt werden sollen. Derartige Beschränkungen, beispielsweise Auslieferungseigenschaften oder Lebenszeit, sind unabhängig von konkreten Signalparametern, werden also dem Signaltyp zugeordnet.

HERKUNFTSQUELLE. Das Konzept Signaltyp in *CoRE$_{CEPT}$* basiert auf RM-ODP. Die Aufnahme von Signalparameter und die Trennung zwischen Signaltyp und Signalparameter wurde in *CoRE$_{CEPT}$* eingeführt.

3.1.11 Medientyp, Medium und Medienmenge

CHARAKTERISIERUNG. Medientyp, Medium und Medienmenge sind Instanzen von Klasse. Das Konzept Medienmenge bildet die Grundlage von *Continuous-Media*-Interaktionen, d.h. des unidirektionalen, kontinuierlichen Austausches potentiell unendlicher Datenströme. Das Konzept Medienmenge aggregiert ein oder mehrere Medien, wobei der Austausch eines Mediums zwischen einem Produzenten und einem Konsumenten (COs) einen atomaren Datenstrom bildet. Zur Modellierung konkreter multimedialer Information, z.B. eines Films oder eines Musikstücks, wird das Konzept Medium in $CoRE_{CEPT}$ aufgenommen. Der Austausch (d.h. die Codierung, die Übertragung und Decodierung des Datenstromes) eines konkretes Mediums erfolgt anhand genau eines Medientyps entsprechend [RFC MIME], wobei zur Ausführungszeit während der Initialisierungsphase unter potentiell unterschiedlichen Medientypen gewählt werden kann.

MOTIVATION. Die Aggregation von Medien innerhalb einer Medienmenge dient zum einen dem Zwecke der Strukturierung, zum anderen der Modellierung gemeinsamer Aspekte wie Synchronisation, Aktivierung und Deaktivierung.

Die Trennung zwischen Medien und den sie realisierenden Medientypen ist darin begründet, daß eine Realisierung der Medientypen selbst unabhängig von konkreten Medien ist, und demzufolge in separaten Entwurfsmodellen beschrieben und in wiederverwendbaren Softwarekomponenten realisiert werden kann. Die Bereitstellung der Mechanismen zur oben erwähnten Initialisierungsphase sind Bestandteil einer Komponentenarchitektur, die diese Art der Interaktion unterstützt (vgl. [TTK+ 00][TTK+ 00a][KT 99]). Damit ist es Aufgabe der konkreten Ableitungsregeln für diese Architektur, die entsprechenden Mechanismen zu berücksichtigen.

HERKUNFTSQUELLE. RM-ODP und ITU-ODL definieren das Konzept *Stream*-Interface. Diese Definitionen wurden in $CoRE_{CEPT}$ konkretisiert.

3.1.12 Consume-, Produce-, Source- und Sink-Definition

CHARAKTERISIERUNG. *Consume*-, *Produce*-, *Source*- und *Sink*-Definition sind Instanzen von Klasse. Die Konzepte Signaltyp und Medienmenge sind die elementaren Bestandteile der Interaktionsarten Signal- bzw. *Continuous-Media*-Interaktion - vergleichbar mit Parametern von Operationen im Falle operationaler Interaktion. Im Gegensatz zu operationaler Interaktion wird bei Signal- und *Continuous-Media*-Interaktion im Kontext eines Interfacetyps unterschieden zwischen den Rollen, die die Instanz eines Interfacetyps in Bezug auf die Kommunktionsrichtung von Signaltypen bzw. Medienmengen einnimmt. Für einen konkreten Signaltyp kann der potentielle Empfang von Signalen dieses Typs durch das Konzept der *Consume*-Definition modelliert werden. In der Gegenrichtung wird das Versenden von Signalen dieses Typs durch das Konzept der *Produce*-Definition dargestellt. Die Konzepte *consume* und *produce* werden durch Relationen zwischen Signaltypen und Interfacetypen in einem Entwurfsmodell identifizierbar (d.h. durch Namensgebung) erfaßt. Analog zu *consume* kann der Empfang einer Medienmenge durch eine *Sink*-Relation modelliert werden, das Versenden einer Medienmenge durch eine *Source*-Definition.

MOTIVATION. Die Instanziierung der Konzepte *Consume*-, *Produce*-, *Source*- und *Sink*-Definition gestattet - wie die Instanziierung von Operation - die Beschreibung von Informationsmanipulation. Konkret wird damit die Modellierung der Signal- bzw. *Continuous-Media*-Interaktion zwischen COs ermöglicht.

HERKUNFTSQUELLE. RM-ODP definiert bereits die Produktion bzw. Konsumption eines Signals sowie Quelle und Senke eines Datenstromes. Allerdings werden dafür keine eigenständigen Konzepte eingeführt. Eine diesbezügliche Präzisierung wurde durch die Aufnahme von *Consume*-, *Produce*-, *Source*- und *Sink*-Definition in $CoRE_{CEPT}$ erreicht.

3.1.13 Interaktionselement

CHARAKTERISIERUNG. Interaktionselement ist Instanz von Klasse. Es werden Generalisierungsbeziehungen zwischen Interaktionselement und *Consume*-, *Produce*-, *Source*- und *Sink*-Definition sowie Operation definiert. Des weiteren wird eine Aggregation zwischen Interfacetyp und Interaktionselement definiert (vgl. Abb.13).

MOTIVATION. Mit der Einführung der Konzepte *Consume*- und *Produce*- sowie *Source*- und *Sink*-Definition sind die Voraussetzungen zur Erweiterung des Konzeptes Interfacetyp um die Interaktionsarten *Continuous-Media*- und Signalinteraktion geschaffen. Konzeptionell sind *Consume*- und *Produce*- sowie *Source*- und *Sink*-Definition im Sinne von elementaren Interaktionen äquivalent mit dem Konzept Operation. Aus diesem Grunde werden alle diese Konzepte zusammengefaßt und als Generalisierung Interaktionselement in *CoRE$_{CEPT}$* aufgenommen. In Entwurfsmodellen, die auf *CoRE$_{CEPT}$* basieren, können Interfacetypen modelliert werden, die die drei Interaktionsarten operationale, *Continuous-Media*- und Signalinteraktion integrieren.

HERKUNFTSQUELLE. Interaktionselement ist ein in *CoRE$_{CEPT}$* neu definiertes Konzept.

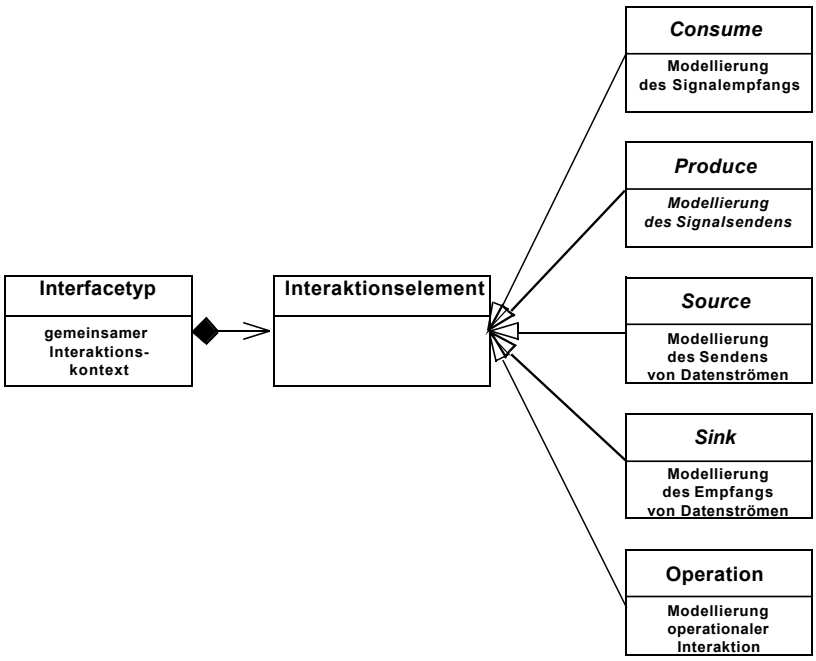


Abb. 13 Interaktionskontext und Interaktionselemente

3.1.14 Diskussion

Offensichtlich beinhaltet der in *CoRE$_{CEPT}$* gewählte Ansatz der Kombination der unterschiedlichen Interaktionsarten im Kontext eines Interfacetyps den Vorteil, daß die Konzepte CO-Typ und *port* sowie die bereits definierten Relationen zum Konzept Interfacetyp unberührt bleiben. Die damit verbundenen technologischen Implikationen werden durch die automatische Ableitung von Softwarekomponenten vollständig abgedeckt, obwohl keine derzeit verfügbare Komponentenarchitektur eine Integration der drei Interaktionsarten im Kontext eines Interfacetyps *a priori* unterstützt. Die Einführung einer solchen Komponentenarchitektur hätte natürlich keinen Einfluß auf existierende Entwurfsmodelle, die auf der Basis von *CoRE$_{CEPT}$* entworfen wurden. Vielmehr würden durch derartige Architekturen die Ableitungsregeln für Softwarekomponenten aus Entwurfsmodellen vereinfacht.

3.1.15 Artefakt

CHARAKTERISIERUNG. Artefakt ist Instanz von Klasse. Instanzen von Artefakt sind Abstraktionen spezifischer, instanziierbarer, programmiersprachlicher Konstrukte (z.B. von Klassen einer objektorientierten Programmiersprache). Instanzen dieser Konstrukte realisieren das Verhalten, die Identität und den Zustand von COs.

MOTIVATION. Softwarekomponenten, die aus auf *CORE*$_{CEPT}$ basierenden Entwurfsmodellen abgeleitet werden, realisieren das Verhalten von COs und damit den Interaktionselementen an den unterstützten bzw. benutzten Interfacetypen. Wie in Abschnitt 1.4 bereits eingeführt, bildet das Konzept Artefakt die Grundlage der Modellierung von programmiersprachlichen Konstrukten zur Realisierung des Verhaltens von COs. Die Struktur einer Implementierung zur Verhaltensrealisierung, d.h. die modellierten Artefakte und ihre Relationen, sowie die Zusammenhänge zwischen Artefakten und Interaktionselementen von Interfacetypen im Kontext eines CO-Typs werden in das Entwurfsmodell aufgenommen.

- Die Entwurfsmodellierung dieser Aspekte eröffnet - unabhängig von einer konkreten Programmiersprache - die Möglichkeit zur parametrisierten Wiederverwendung einer bewährten Implementierungsstrategie.
- Durch die Modellierung von Artefakten können Relationen zwischen neu zu entwerfenden Systemkomponenten und bestehenden Implementierungspaketen leicht erfaßt und durch automatische Ableitung dieser Systemkomponenten hergestellt werden.
- Anforderungen an Softwarekomponenten wie z.B. Skalierbarkeit lassen sich durch die Modellierung allgemeiner Muster (*policies*) für Artefakte und ihre Erzeugung bzw. Zerstörung explizit im Entwurfsmodell erfassen - wiederum unabhängig von einer konkreten Programmiersprache.

HERKUNFTSQUELLE. Artefakt verallgemeinert die Konzepte von *Component Implementation Framework* aus dem CORBA-Komponentenmodell - das CORBA-Komponentenmodell sieht eine Definition der Strukturierung der Implementierung einer CORBA-Komponente in Segmente vor.

3.1.16 Implementierungselement, Zustandsattribut und Implements-Relation

CHARAKTERISIERUNG. Implementierungselement ist Instanz von Klasse. Das Konzept Implementierungselement erfaßt diesen Zusammenhang zwischen Interaktionselement eines Interfacetyps und programmiersprachlicher Realisierung dieses Interaktionselements im Kontext von Artefakten. Es ist eine Assoziation zwischen Implementierungselement und Interaktionselement definiert. Artefakt aggregiert Implementierungselement. Zustandsattribut ist Instanz von Klasse, es ist eine Assoziation von Zustandsattribut zum Konzept Datentyp definiert. CO-Typ aggregiert Zustandsattribut.

Zustandsattribute werden im Kontext von CO-Typen definiert, um Implementierungskonstrukten (Artefakten und von diesen aggregierten Implementierungselementen) für CO-Typen Zustandsinformationen von COs zugänglich zu machen. Die Definition eines Implementierungselements in einem Entwurfsmodell erfolgt mit der Semantik, daß eine Instanz des Artefakts, das ein Implementierungselement aggregiert, für die Erbringung des Verhaltens des diesem Implementierungselement zugeordneten Interaktionselements verantwortlich ist. Um die Menge derjenigen Artefakte bestimmen zu können, die das Verhalten eines konkreten CO-Typs realsieren, muß in einem Entwurfsmodell eine Zuordnung zwischen Artefakten und CO-Typen hergestellt werden. Die konzeptionelle Basis dafür wird durch die Einführung der *Implements*-Relation geschaffen. Sie ist als Assoziation zwischen den Konzepten CO-Typ und Artefakt definiert.

MOTIVATION. Um mittels des Konzeptes Artefakt die Realisierung des Verhaltens von COs zu modellieren, wird das Konzept Implementierungselement als Verbindung zwischen an Interfacetypen definierten Interak-

tionselementen und Artefakten eingeführt. Mit der Einführung des Konzeptes Artefakt sind zunächst programmiersprachliche Konstrukte in Entwurfsmodellen erfaßbar, die das Verhalten von CO-Typen realisieren. Der wesentliche Teil dieser Verhaltensrealisierung ist die Verarbeitung von Interaktionselementen, z.B. die Implementierung einer an einem Interfacetyp angebotenen Operation oder das Verhalten beim Empfang eines Signals.

Der Zustand eines COs drückt sich in der Belegung von *internen*, nach außen *nicht notwendigerweise sichtbaren* Attributen aus. Diese Attribute werden hier als Zustandsattribute bezeichnet. Der Zugriff auf diese Zustandsattribute bezüglich des durch ein Artefakt realisierten CO-Typs ist für die korrekte Verhaltensrealisierung notwendig. Zu diesem Zweck wird hier das Konzept Zustandsattribut eingeführt, eine Assoziation zum Konzept Datentyp definiert und zu Implementierungselementen assoziiert. Die Zusammenhänge zwischen den Konzepten Implementierungselement, Zustandsattribut und Interaktionselement sind in Abb. 14 illustriert.

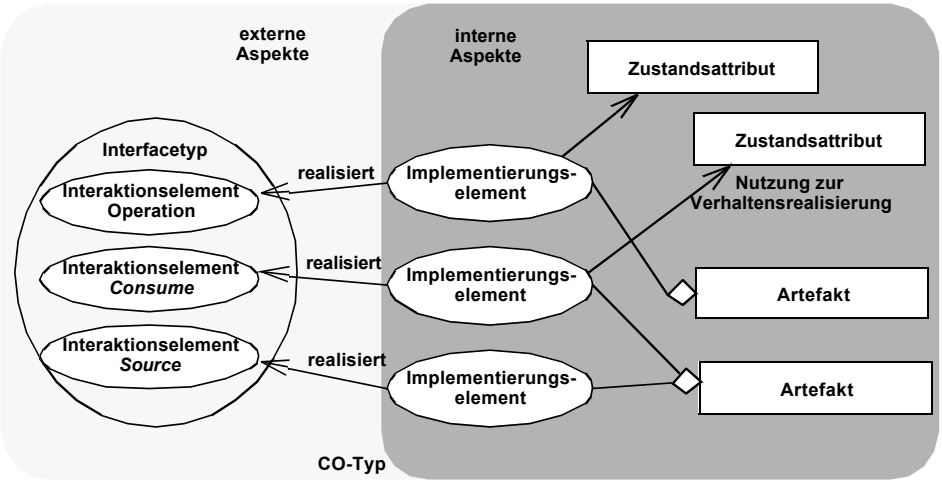


Abb. 14 Interaktionselemente, Artefakte und Implementierungselemente im Kontext eines CO-Typs

HERKUNFTSQUELLE. Die konzeptionelle Grundlage von *CoRE$_{CEPT}$* für Artefakt, Zustandsattribut und Implementierungselement ist das CORBA-Komponentenmodell. Die Modellierung der Verhaltensrealisierung von CO-Typen wurde dort bereits diskutiert. Die Strukturierungs- und Zuordnungsmittel zwischen den Interaktions- und den Implementierungselementen sind aber stark eingeschränkt. So müssen alle Interaktionselemente im Kontext eines Interfacetypen durch ein und dasselbe Artefakt (in [OMG CCM I] als Segment bezeichnet) realisiert werden. Die in [OMG CCM I] enthaltenen Ansätze wurden in *CoRE$_{CEPT}$* verallgemeinert.

3.1.17 Instanziierungsmuster

CHARAKTERISIERUNG. Instanziierungsmuster beschreiben, nach welchen Regeln während der Ausführung einer Softwarekomponente auf einer Maschine Instanzen der verschiedenen, durch Artefakte modellierten Implementierungskonstrukte angelegt werden. Instanziierungsmuster für Artefakte werden in *CoRE$_{CEPT}$* durch einen Aufzählungstyp beschrieben. Es sind die folgenden Elemente dieses Aufzählungstyps definiert: *ARTIFACT_PER_REQUEST, ARTIFACT_POOL, SINGLETON* und *USER_DEFINED.*

MOTIVATION. Die Entkopplung der Identität von COs von der Identität der Instanzen von programmiersprachlichen Artefaktrepräsentationen (Artefaktinstanzen) erlaubt die Anwendung unterschiedlichster

Instanziierungsmuster für Artefakte (vgl. Abb. 15 in UML-Notation). Beispielsweise kann eine Artefaktinstanz das Verhalten einer Menge von COs realisieren (Artefaktinstanz per CO-Typ), eine Menge von Artefaktinstanzen kann das Verhalten einer Menge von COs realisieren (Artefaktinstanz-*Pool* per CO-Typ) oder eine Artefaktinstanz kann das Verhalten exakt eines COs realisieren (Artefaktinstanz per CO). Insbesondere kann ein und dieselbe Artefaktinstanz zur Implementierung des Verhaltens von COs unterschiedlichen Typs genutzt werden (Artefakt *sharing*). Eine geeignete, an den Einsatzweck angepaßte Verwendung von Instanziierungsmustern im Entwurf eines Softwaresystems sichert hohe Performanz - die Eignung konkreter Instanziierungsmuster für Artefakte kann durch den Entwickler simulativ oder experimentell ermittelt werden. Diese verschiedenen Instanziierungsmuster sind in $CoRE_{CEPT}$ erfaßt:

- *ARTIFACT_PER_REQUEST* - für jeden Zugriff auf ein Implementierungselement des Artefakts, d.h. beim Zustandekommen der Interaktion bezüglich des zugordneten Interaktionselements, wird eine neue Artefaktinstanz erzeugt,
- *ARTIFACT_POOL* - eine fixe Anzahl von Artefaktinstanzen wird vor dem Zustandekommen von Interaktionen erzeugt, beim Zugriff auf ein zugehöriges Implementierungselement wird eine Artefaktinstanz aus dieser Menge ausgewählt,
- *SINGLETON* - genau eine Artefaktinstanz wird initial erzeugt, jeder Zugriff auf Implementierungselemente des Artefakts wird durch diese Instanz realisiert,
- *USER_DEFINED* - der programmiersprachliche Mechanismus zur Erzeugung von Artefakten wird durch den Entwickler bereitgestellt und daher nicht durch die Ableitungsregeln für die automatische Ableitung von Softwarekomponenten berücksichtigt.

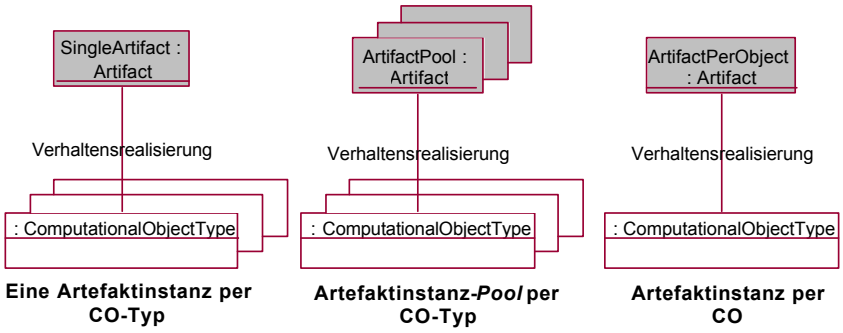


Abb. 15 Instanziierungsmuster für Artefakte

HERKUNFTSQUELLE. Die Definition von Instanziierungsmustern für Artefakte wurde in $CoRE_{CEPT}$ eingeführt.

3.1.18 Diskussion

Durch die Trennung der Konzepte CO-Typ und Artefakt sowie Interaktions- und Implementierungselement ergibt sich eine hohe Flexibilität der Komposition von Artefakten zur Implementierung des Verhaltens eines oder mehrerer CO-Typen (*Komponierbarkeitsflexibilität*). Flexible Komponierbarkeit ist immer dann gegeben, wenn ein Artefakt nicht das gesamte erwartete Verhalten eines CO-Typs implementiert, sondern eventuell nur einen Teil desselben (vgl. Abb.16).

Darüber hinaus ergibt sich die Möglichkeit, unterschiedliche Verfahren der Projektion von COs auf Artefaktinstanzen durch Instanziierungsmuster auszunutzen (*Instanziierungsflexibilität*). Die adäquate Umsetzung von in einem auf $CoRE_{CEPT}$ basierenden Entwurfsmodell definierten Instanziierungsmustern für Artefakte konkreter Softwarekomponenten ist Bestandteil der automatischen Ableitung dieser Komponenten.

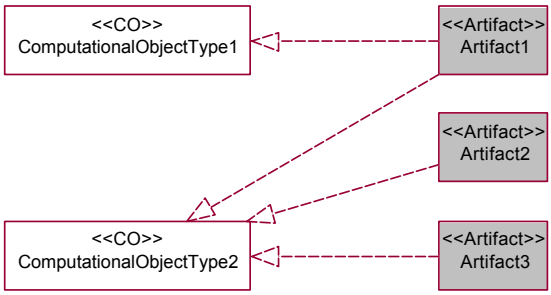


Abb. 16 Realisierung von CO-Verhalten durch Artefakte

In *CoRE$_{CEPT}$* wird die konkrete Realisierung der Implementierungselemente durch Artefakte nicht im Entwurfsmodell erfaßt, muß also in den programmiersprachlichen Konstrukten, die für ein Artefakt durch *CoRE$_{MAP}$* erzeugt werden, vorgenommen werden. Zum gegewärtigen Zeitpunkt ist diese Vorgehensweise sinnvoll: Gerade zur Beschreibung funktionaler Aspekte - des systemspezifischen Verhaltens („*Business Logic*“) - für konkrete Implementierungselemente sind programmiersprachliche Mittel adäquat. Falls sich eine - bisher nicht existierende - technologische Lösung des im Dokument „*Action Semantics for the UML Request For Proposal*“ [OMG AS RFP] vorgestellten Ansatzes einer plattform- und programmiersprachenunabhängigen Repräsentation programmiersprachlicher Mittel in Entwurfsmodellen in der Praxis bewährt, so kann genau dieser Ansatz zur Modellierung der Realisierung von Implementierungselementen durch Artefakte in *CoRE$_{CEPT}$* integriert werden. Ebenso kann eine Integration formaler Beschreibungstechniken, wie z.B. [ITUT Z.100], vorgenommen werden. Voraussetzung für die Überführung solcher Beschreibungen in Softwarekomponenten ist dann die Definition geeigneter Ableitungsregeln für eine Repräsentation in Programmiersprachen.

Die bisher eingeführten Konzepte erlauben die Modellierung struktureller Aspekte von CO-Typen, deren potentielle Konfiguration durch Beschreibung von *Port*-Definitionen sowie die Definition der Implementierung durch Artefakte. Weiterhin können konkrete Relationen zwischen der Struktur von CO-Typen und der Implementierungsstruktur sowie Aussagen bezüglich bestimmter Verhaltensaspekte wie Lebenszeit und Instanziierungsmuster für Artefakte erfaßt werden.

Der folgerichtige Schritt ist nun die Definition der Relationen zwischen CO-Typen und den sie realisierenden Artefakten auf der einen sowie der diese realisierenden Softwarekomponente auf der anderen Seite. Anschaulich bedeutet diese Relation, daß durch die in einer Softwarekomponente enthaltenen Codemodule während ihrer Ausführung auf einer Maschine COs eines oder mehrerer konkreter CO-Typen erzeugt werden können. Die Repräsentation der COs entspricht gerade den im Entwurfsmodell angegebenen Informationen.

3.1.19 Softwarekomponente und Realize-Relation

CHARAKTERISIERUNG. Softwarekomponente ist Instanz von Klasse, die *Realize*-Relation wird als Aggregationsbeziehung von Softwarekomponente zu CO-Typ in *CoRE$_{CEPT}$* definiert.

Das Konzept Softwarekomponente dient zur Modellierung von Softwarekomponenten, wobei von den tatsächlichen, in Softwarekomponenten enthaltenen Instruktionssequenzen abstrahiert wird. Softwarekomponenten realisieren einen oder mehrere CO-Typen in dem Sinne, daß sie die Instruktionssequenzen beinhalten, die Zustand, Identität und Verhalten von Objekten implementieren, die COs während der Ausführungszeit repräsentieren. Die Zuordnung von modellierten CO-Typen (Instanzen des Konzeptes CO-Typ) zu modellierten Softwarekomponenten (Instanzen des Konzeptes Softwarekomponente) wird durch das Konzept *Realize*-Relation erfaßt.

MOTIVATION. Eine Softwarekomponente enthält Instruktionssequenzen, d.h. diejenigen programmiersprachlichen Konstrukte, die durch Artefakte im Entwurfsmodell beschrieben sind, sowie durch die automatische Ableitung der Softwarekomponente produzierte programmiersprachliche Konstrukte, die die realisierten CO-Typen repräsentieren. Dabei werden solche Artefakte berücksichtigt, deren Instanzen das Verhalten, den Zustand und die Identität von COs repräsentieren, die der Softwarekomponente zugeordnet wurden. Diese Zuordnung wird mit einer *Realize*-Relation im Entwurfsmodell erfaßt (vgl. Abb. 17). Der Grund für die Aufnahme von Softwarekomponenten in Entwurfsmodelle besteht darin, die Grundlage für die automatische Ableitung von Softwarekomponenten zu schaffen: Modelle von Softwarekomponenten definieren die Ziele der automatischen Ableitung von Softwarekomponenten eines Softwaresystems.

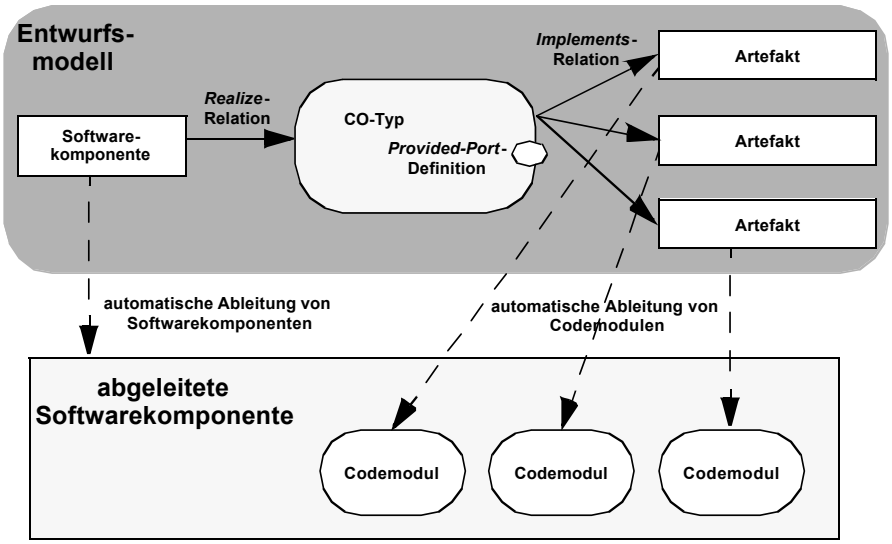


Abb. 17 Ableitung von Softwarekomponenten aus Entwurfsmodellen

HERKUNFTSQUELLE. Das Konzept Softwarekomponente und die *Realize*-Relation in $CORE_{CEPT}$ basieren auf *Packaging and Deployment* des CORBA-Komponentenmodells sowie auf den Resultaten des EURESCOM-Projektes P924 [EU P924], an dem die Autoren aktiv beteiligt sind.

3.1.20 Assemblage und initiale Konfiguration

CHARAKTERISIERUNG. Das Konzept Assemblage ist Instanz von Klasse. Die Konzepte CO (Instanz von CO-Typ) und Bindung sind Instanzen von Klasse. Die Verbindung zum Konzept Assemblage wird durch Instanzen von Aggregation und Assoziation hergestellt.

Das Konzept Assemblage wird eingesetzt, um alle zu einem verteilten Softwaresystem gehörigen CO-Typen und deren initiale Konfiguration im Entwurfsmodell zu erfassen. Instanzen sind Modell eines verteilten Softwaresystems, wobei auf die funktionale Dekomposition des Softwaresystems, also auf die CO-Typen und die beabsichtigte initiale Konfiguration fokussiert wird. Die initiale Konfiguration ist definiert durch eine Menge von Instanzen von CO-Typen (den initialen COs) sowie einer Menge von Bindungen (initialen Bindungen) zwischen *Port*-Definitionen der zu den jeweiligen Instanzen gehörigen CO-Typen. Eine Bindung darf zwischen *Port*-Definitionen definiert werden, wenn mindestens eine der beteiligten *Port*-Definitionen eine *Used-Port*-Definition und eine weitere eine *Provided-Port*-Definition ist. Die Semantik einer Bindung besteht in dem Austausch der Referenzen der zu den *Port*-Definitionen gehöhrenden Interfaces unter Beachtung der Art der *Port*-Definition (*Used*- oder *Provided-Port*-Definition) zum Zeitpunkt des Aufbaus der initialen Konfiguration.

MOTIVATION. Ein verteiltes Softwaresystem besteht aus einer Menge von Softwarekomponenten, die auf Maschinen bereitgestellt und installiert werden. Programmiersprachliche Objekte, die COs repräsentieren, werden während der Ausführung einer Softwarekomponente auf der Grundlage der Beschreibung des CO-Typs instanziiert. Die Instanziierung ist Teil des Verhaltens des Softwaresystems und wird von anderen COs oder Managementmechanismen veranlaßt. Als problematisch hat sich in der Vergangenheit die Erzeugung der initialen Instanzen von CO-Typen (initiale COs) sowie der Austausch von Interfacereferenzen zwischen diesen herausgestellt. Es besteht der Wunsch [EU P924][EUP924a], die initiale Konfiguration eines verteilten Systems im Entwurfsmodell zu erfassen und auf der Grundlage dieser Beschreibung die tatsächliche Konfiguration während des Einsatzes des Softwaresystems durch geeignete Managementmechanismen der genutzen Komponentenarchitektur aufbauen zu lassen. *CoRE$_{CEPT}$* wird dieser Anforderung durch die Konzepte Assemblage und initiale Konfiguration gerecht.

HERKUNFTSQUELLE. Die Konzepte Assemblage und initiale Konfiguration in *CoRE$_{CEPT}$* basieren auf *Packaging and Deployment* des CORBA-Komponentenmodells sowie auf den Resultaten des EURESCOM-Projektes P924.

3.1.21 Diskussion

Deployment eines verteilten Softwaresystems erfolgt auf der Grundlage einer im Entwurfsmodell enthaltenen Assemblagedefinitionen für dieses System. Dabei müssen jedoch einige Regeln beachtet werden. So können Softwarekomponenten i.allg. nicht auf jeder beliebigen Maschine ausgeführt werden, sondern nur auf einer für die jeweilige Komponente geeigneten. Ist beispielsweise eine Komponente für das Betriebssystem *Microsoft Windows* erstellt worden, so ist die Präsenz dieses Betriebssystem auf der für die Ausführung vorgesehenen Maschine erforderlich. Weitere Regeln können sich z.B. auf bestimmte Ausstattungsmerkmale der Maschine wie Hauptspeicher oder Verarbeitungsgeschwindigkeit beziehen. Um eine Automatisierung von *Deployment* unter Beachtung solcher Regeln zu erreichen, werden auch die Eigenschaften von Softwarekomponenten und Regeln zu ihrer Verteilung im Entwurfsmodell erfaßt. Ein Ansatz für die Modellierung dieser Information wird z.Zt. innerhalb eines EURESCOM Projektes [EU P924] unter Beteiligung der Autoren erarbeitet. Eine weitergehende Integration der Projektergebnisse in *CoRE* ist vorgesehen.

Mit den bisher eingeführten Konzepten ist die Modellierung der Struktur von CO-Typen, deren Realisierung durch Artefakte sowie die Zusammenfassung von CO-Typen in realisierende Komponenten möglich. Zusätzlich können initiale Konfigurationen von COs und ihren Bindungen beschrieben werden.

Das Konzept von Bindungen wird im folgenden erweitert, um die Modellierung der Eigenschaften dieser Bindungen zu gestatten (Bindungsregeln). Insbesondere sollten nicht nur die Bindungen beschreibbar sein, die in einer initialen Konfiguration auftreten, sondern vielmehr eine allgemeine Modellierung von Bindungscharakteristika unter Beachtung modellierter Umgebungskonstellationen ermöglicht werden. Bindungseigenschaften und ihre Erfassung in Entwurfsmodellen sind die Grundlage für die Umsetzung der Anforderung nach Gütebeschreibung und -garantie. Die Modellierung von Bindungseigenschaften erfolgt fallbasiert - wie durch das folgende Beispiel illustriert.

(Beispiel 10) Ein CO-Typ, der den Zugriff auf spezifische Daten eines Unternehmens ermöglicht, stellt diese Funktionalität an einem unterstützten Interface bereit. Klienten, die dieses Interface nutzen, können auf Maschinen ausgeführt werden, die räumlich verteilt sind. Bei der Bindung wird der Standort der Klientenmaschine als Umgebungskonstellation berücksichtigt (vgl. Abb. 18):

- Sofern die Klientenmaschine zum geschützten *Intranet* des Unternehmens gehört, sind keine Verschlüsselungsmaßnahmen für die Bindung anzuwenden,
- sofern die Klientenmaschine jedoch dem *Extranet* des Unternehmens (d.h. dem Teil des Internet, der nicht zum Intranet gehört) zuzuordnen ist, soll beispielsweise *Secure Socket Layer* (SSL, [Sun SSL]) zur Übertragung benutzt werden. Diese Forderung wird als Eigenschaft der Bindung eines Klienten an COs im Entwurfsmodell erfaßt.

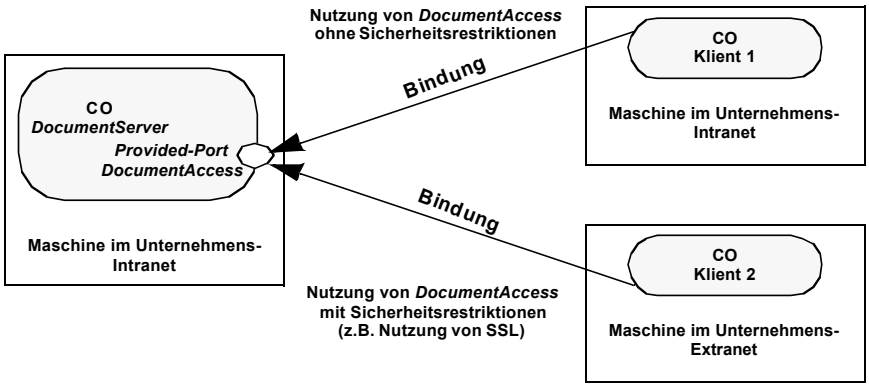


Abb. 18 Bindungsrestriktionen in Abhängigkeit von Umgebungskonstellationen

Zur Modellierung der genannten Aspekte wird $CoRE_{CEPT}$ im folgenden erweitert.

3.1.22 Bindung mit Regel und Kontrakttyp

CHARAKTERISIERUNG. Das Konzept Kontrakttyp in $CoRE_{CEPT}$ ist Instanz von Klasse. Das Konzept Bindung (ebenfalls Klasse in $CoRE_{CEPT}$) wird spezialisiert durch das Konzept Bindung mit Regel (Bindungsregel). Bindungsregeln sind logische Ausdrücke, die sich auf Attribute spezieller Datentypen - den Kontrakttypen - beziehen. Sie dienen zur Modellierung der gewünschten, d.h. zur Ausführungszeit zu gewährleistenden Eigenschaften von Bindungen.

Die Modellierung der im Rahmen einer Bindungsregel zulässigen Kontrakttypen und damit der verwendbaren Attribute wird durch die Definition einer Assoziation zwischen Interfacetyp und Kontrakttyp in $CoRE_{CEPT}$ ermöglicht. Nur Attribute von konkreten Kontrakttypen in Entwurfsmodellen, für die eine Relation zu einem Interfacetyp modelliert wurde, dürfen in Bindungsregeln verwendet werden, an denen dieser Interfacetyp beteiligt ist.

MOTIVATION. Die Spezifikation von Bindungen mit Regeln ist essentiell zur Umsetzung der Anforderung nach Gütebeschreibung und -garantie. Nur wenn die Anforderungen im Sinne von umzusetzenden Regeln im Entwurfsmodell erfaßt sind, können Softwarekomponenten durch $CoRE_{MAP}$ automatisch erzeugt werden, die in der Lage sind, im Zusammenwirken mit $CoRE_{WARE}$ die Umsetzung dieser Anforderungen zur Ausführungszeit zu gewährleisten.
Sind die Bindungsregeln auf der Basis von Kontrakttypen formuliert, so erleichtert sich deren Auswertung, da die Attribute und ihre Typen im Vorfeld (d.h. zum Zeitpunkt der Ableitung der Softwarekomponenten) bekannt sind. Die Vorteile von typbasierten Beschreibungen, wie höhere Aussagekraft von Entwurfsmodellen und bessere Möglichkeiten zu ihrer Überprüfung, lassen sich durch das Konzept Kontrakttyp auf die Beschreibung von Bindungsregeln übertragen.

HERKUNFTSQUELLE. Das Konzept Bindung mit Regel konkretisiert RM-ODP. Das Konzept Kontrakttyp ist einer verbreiteten Notation zur Modellierung von *Quality-of-Service*-Eigenschaften entnommen [FTK+ 00].

3.1.23 Bindungsfall und Prädikat

CHARAKTERISIERUNG. Bindungsfall und Prädikat sind Instanzen von Klasse. Die Beziehung zwischen CO-Typ und Prädikat wird durch eine Assoziation hergestellt.

Ein Bindungsfall beinhaltet eine Menge von COs, Bindungen dieser COs und zu diesen Bindungen gehörenden Bindungsregeln. Er beschreibt, unter welcher Umgebungskonstellation die durch die Bindungsregeln beschriebenen Anforderungen zur Ausführungszeit zu realisieren sind. Die Charakterisierung der COs bzw. der CO-Instanzmengen, die an einem spezifischen Bindungsfall beteiligt sind, erfolgt durch Angabe von Prädikaten. Instanzen des Konzeptes Prädikat sind Abstraktionen von zu wahr oder falsch evaluierbaren Ausdrücken im Kontext eines CO-Typs. Die Auswertung der Prädikate eines Bindungsfalles zur Ausführungszeit führt zur Identifizierung der Umgebungskonstellation, für den dieser Bindungsfall anzuwenden ist.

MOTIVATION. Die Interaktion zwischen COs muß i.allg. speziellen Anforderungen hinsichtlich der Güte und den Eigenschaften der Interaktion genügen. Dabei hängen die zu realisierenden Anforderungen nicht ausschließlich von den beteiligten COs ab, sondern auch davon, unter welchen Konstellationen die Kommunikation zustande kommt. Daher müssen auch Beschreibungen von Umgebungskonstellationen Bestandteil von Entwurfsmodellen sein. Zu diesem Zweck dienen die Konzepte Bindungsfall und Prädikat, wobei über die Angabe von Prädikaten die Umgebungskonstellation, unter der der Bindungsfall zur Anwendung kommt, bestimmt wird. Ein Bindungsfall beinhaltet wiederum eine Menge von Bindungsregeln, die dann zu gewährleisten sind.

HERKUNFTSQUELLE. Bindungsfall und Prädikat sind in *CoRE$_{CEPT}$* neu definierte Konzepte.

3.1.24 Diskussion

Mit den eingeführten Konzepten ist sowohl die Modellierung von strukturellen Aspekten von interagierenden COs sowie ihre Realisierung durch Softwarekomponenten als auch von Verhaltensaspekten bezüglich der potentiellen Bindungen zwischen diesen COs möglich.

Die Modellierung dieser Verhaltensaspekte wie auch der auf Instanziierungsmustern basierende Ansatz zur Modellierung von Eigenschaften der Relationen zwischen Implementierungselementen und Artefakten erfolgt auf der Grundlage der innerhalb des Entwicklungsprozesses definierten Anwendungsfälle des Softwaresystems. Die Modellierung dieser Anwendungsfälle ist Bestandteil der Anforderungsdefinition, die in der Analyse- und Definitionsphase erarbeitet wird. Es wird davon ausgegangen, daß die Anforderungsdefinition bereits vorliegt und zur Modellierung von Bindungsfällen und Instanziierungsmustern herangezogen werden kann.

3.2 Systematisierung von Konzepten in CoRE$_{CEPT}$

Um eine übersichtliche Strukturierung von Modellen zu erreichen - insbesondere zur Ermöglichung einer verbesserten Werkzeugunterstützung - ist die Separierung und Ordnung von Modellierungsaspekten ein

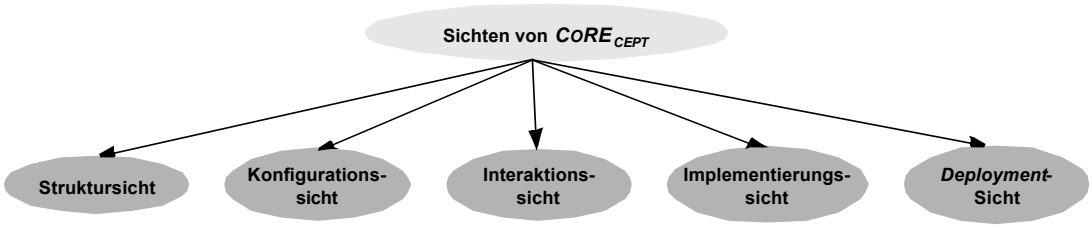


Abb. 19 Einordnung der Konzepte von CoRE$_{CEPT}$ in Sichten

geeignetes Mittel. Dieses wurde ebenfalls in RM-ODP angewendet. Für $CORE_{CEPT}$ ergibt sich eine natürliche Einteilung der Konzepte wie folgt (vgl. Abb. 19):

- *Struktursicht* - strukturelle Aspekte bezüglich CO-Typen und ihren durch Interfacetypen beschriebenen potentiellen Interaktionen,
- *Konfigurationssicht* - Aspekte der Konfiguration von COs,
- *Implementierungssicht* - Aspekte der Realisierung des Verhaltens von COs durch Artefakte,
- *Deployment-Sicht* - Aspekte der Modellierung von Softwarekomponenten und ihrer Eigenschaften,
- *Interaktionssicht* - Beschreibung der Eigenschaften von potentiellen Bindungen zwischen COs.

Die genaue Zuordnung der Konzepte von $CORE_{CEPT}$ in die definierten Sichten ist in Tab. 1 dargestellt:

Sicht	Enthaltene Konzepte
Struktursicht	Namensraum, Datentyp, Operation, Ausnahme und Parameter, Signaltyp und Signalparameter, Medientyp, Medium und Medienmenge, *Consume*-, *Produce*-, *Source*- und *Sink*-Definition, Interaktionselement, Interfacetyp, CO-Typ, *Supports*- und *Requires*-Relation
Konfigurationssicht	*Port*-Definition, *Provided*- und *Used-Port*-Definition
Implementierungssicht	Artefakt, Implementierungselement, Zustandsattribut, *Implements*-Relation, Instanziierungsmuster
Deployment-Sicht	Softwarekomponente, *Realize*-Relation, Assemblage, CO und Bindung (initiale Konfiguration)
Interaktionssicht	Bindung mit Regel , Kontrakttyp, Prädikat, Bindungsfall

Tab.1 Sichten und Konzepte von $CORE_{CEPT}$

3.3 Charakterisierung von $CORE_{TATIONS}$

Die Terminus Notation bedeutet die Definition der Art und Weise der Darstellung von Instanzen der Konzepte von $CORE_{CEPT}$. Die Darstellung dieser Instanzen kann dabei graphisch oder textuell erfolgen.

Der in *CORE* gewählte Ansatz der vollständigen Trennung der Beschreibung der zur Modellierung verwendbaren Konstrukte in einem Konzeptraum von einer konkreten Technik zur Notation von Modellen gewährleistet einen hohen Freiheitsgrad bezüglich der Definition einer solchen Notation. So ist es denkbar, eine völlig neue Notation zu entwickeln, eine bereits existierende Notation zu benutzen, eine bereits existierende Notation anzupassen oder sogar mehrere Notationen in Kombination einzusetzen. Kriterien zur Unterstützung der Entscheidungsfindung werden in [CoRE II], Kapitel 5 definiert. Hier soll noch einmal auf die Wichtigkeit der Werkzeugunterstützung für Entwicklungstechniken hingewiesen werden: Es ist nicht nur erforderlich, geeignete Werkzeuge für jede einzelne Entwicklungstechnik bereitzustellen, sondern es ist eine professionelle Unterstützung dieser Werkzeuge notwendig. Dies trifft insbesondere auf die Werkzeugunterstützung der Notation zu, da die Notation von Entwurfsmodellen ein Arbeitsschritt während der Entwurfsphase ist, der interaktiv von Entwicklern ausgeführt wird. Damit besteht ein potentiell hoher Aufwand zur Schulung der Entwickler, wenn eine unbekannte Notation verwendet wird.

Dementsprechend konzentrierten die Autoren ihre Untersuchungen bezüglich $CORE_{TATIONS}$ auf Erweiterungen standardisierter, weit verbreiteter Notationen, für die bereits eine Werkzeugunterstützung existiert, und verzichteten auf die Definition einer gänzlich neuen Notation.

3.4 $CoRE_{WARE}$ und $CoRE_{MAP}$

Basierend auf den Definitionen von $CoRE_{CEPT}$ werden durch $CoRE_{MAP}$ Ableitungsregeln definiert, mit deren Hilfe objektorientierte Modelle eines Softwaresystems in ausführbare Softwarekomponenten überführt werden. Um Softwarekomponenten verteilter Systeme implementieren, integrieren und einsetzen zu können, wird in *CoRE* die Komponentenarchitektur CORBA genutzt. Basierend auf CORBA werden *Deployment*- und Ausführungsunterstützung für die mit $CoRE_{MAP}$ erzeugten Softwarekomponenten entwikkelt ($CoRE_{WARE}$).

In der Literatur [EUP910][EU P715] werden bezüglich Komponentenarchitekturen und *Deployment*- und Ausführungsunterstützung für Softwarekomponenten die Termini Serviceplattform, *Component-Support*-Plattform und *Distributed-Processing*-Umgebung erwähnt, jedoch abweichend oder unpräzise eingeführt. Diese Termini sind auch für *CoRE* grundlegend und werden im folgenden diskutiert.

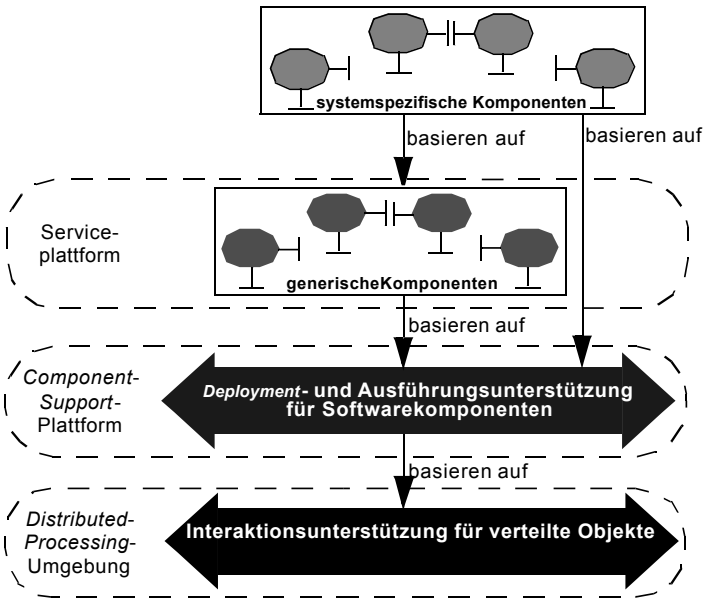


Abb. 20 Distributed-Processing-Umgebung, Komponenten- und Diensteplattform

In $CoRE_{WARE}$ bildet die *Distributed-Processing*-Umgebung die technologische Grundlage zur Unterstützung der Interaktion von Objekten eines verteilten Softwaresystems (s. Abb. 20). Allgemeine Objektdienste - insbesondere zum Auffinden von Objekten - werden ebenfalls durch die *Distributed-Processing*-Umgebung angeboten. Diese Umgebung ist unter Verwendung von CORBA in $CoRE_{WARE}$ realisiert.

Component-Support-Plattformen nutzen die durch die *Distributed-Processing*-Umgebung bereitgestellte Funktionalität, sie bieten die *Deployment*- und Ausführungsunterstützung für Softwarekomponenten an. Während eine *Distributed-Processing*-Umgebung, die beispielsweise unter Nutzung von CORBA-Produkten realisiert ist, nur Objektinteraktion bzgl. eines angebotenen bzw. genutzten Interfaces ermöglicht, stellt eine *Component-Support*-Plattform zur Ausführungszeit den gemeinsamen Kontext aller Interfaces eines COs her. Des weiteren realisiert sie Mechanismen zur Integration, Konfiguration und Inbetriebnahme der Softwarekomponenten. Beispiele für *Component-Support*-Plattformen sind EJB (*Enterprise Java Beans*) und CORBA *Components*, beide können auf einer mit CORBA-2.4-Produkten realisierten *Distributed-Processing*-Umgebung bereitgestellt werden.

Eine Serviceplattform ist konzeptionell eine Menge von vordefinierten, in einem bestimmten Kontext allgemein nutzbaren Softwarekomponenten, die die Dienste der *Component-Support*-Plattform nutzen. Während *Component-Support*-Plattformen weitgehend unabhängig von der konkreten Anwendungsdomäne sind, werden Serviceplattformen domänenspezifisch entworfen (z.B. TINA-Servicearchitektur in der Telekommunikationsdomäne [TINA SA]). Basierend auf durch Serviceplattformen bereitgestellten allgemeinen Softwarekomponenten werden konkrete, anwendungsspezifische Softwarekomponenten definiert.

Ziel der Anwendung von Ableitungsregeln ist die Erzeugung aller Bestandteile, deren Zusammensetzung eine ausführbare Softwarekomponente mit Interfaces für ihren Einsatz ergibt. In Abhängigkeit von der benutzten *Component-Support*-Plattform und der dieser zugrunde liegenden *Distributed-Processing*-Umgebung beinhalten diese Bestandteile:

- *Component-Support*-Plattform-spezifische Interfacebeschreibungen der Softwarekomponenten (abgeleitet aus Interfacetypen und CO-Typen in einem auf *CORE* basierenden objektorientierten Entwurfsmodell),
- programmiersprachliche Konstrukte für die Artefakte eines solchen Modells,
- programmiersprachliche Konstrukte für die Einbindung der Mechanismen der *Component-Support*-Plattform in die entstehenden Softwarekomponenten,
- *Component-Support*-Plattform-spezifische Repräsentation von Bindungsmechanismen (*Policies*, Prädikate und Bindungsregeln),
- *Makefiles* zur Compilation und zum Binden maschinenspezifischer Codemodule (falls erforderlich),
- Deskriptoren für automatisierte *Deployment*-Abläufe.

$CORE_{MAP}$ stellt dabei sicher, daß in einer Softwarekomponente ausschließlich diejenigen Codemodule enthalten sind, die zur Realisierung derjenigen CO-Typen erforderlich sind, für die eine *Realize*-Relation zu dieser Softwarekomponente im Entwurfsmodell vorhanden ist.

3.5 Kombination der Entwicklungstechniken

Die Basis der Kombination der Entwicklungstechniken objektorientierte Modellierung, automatische Ableitung von Softwarekomponenten und Einsatz von Komponentenarchitekturen in *CORE* ist die Definition des Konzeptraumes $CORE_{CEPT}$ (vgl. Abb. 21). Basierend auf diesem Konzeptraum können Notationen defi-

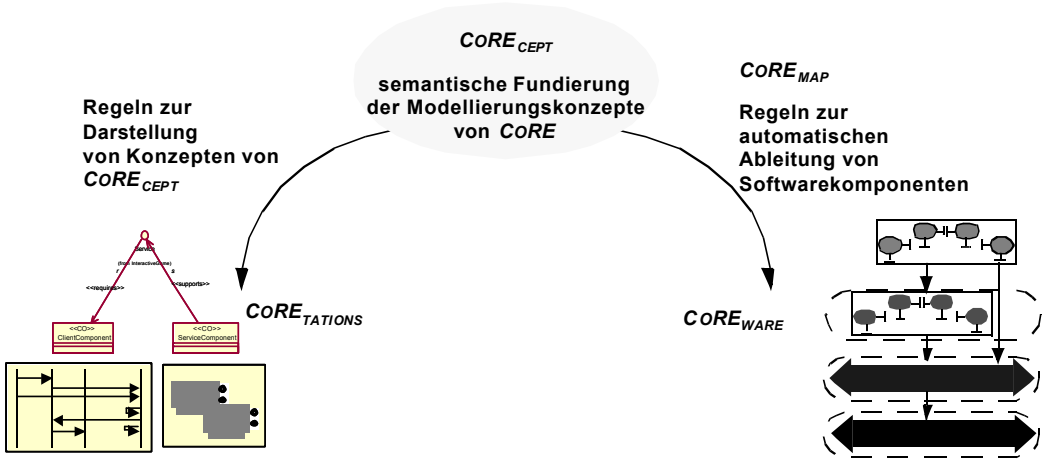


Abb. 21 $CORE_{CEPT}$ als Grundlage der vorgestellten Entwicklungstechniken

niert werden, mit deren Hilfe konkrete objektorientierte Entwurfsmodelle darstellbar sind. Des weiteren erfolgt die Definition der Ableitungsregeln für Softwarekomponenten ebenfalls auf der Grundlage von $CoRE_{CEPT}$.

Die konzeptionelle Trennung zwischen $CoRE_{CEPT}$ und $CoRE_{TATIONS}$ einerseits und $CoRE_{CEPT}$ und $CoRE_{MAP}$ andererseits erweist sich als sehr vielversprechend:

- Notationen, die über einen eigenen Konzeptraum verfügen, lassen sich durch die Definition von Regeln für Transformationen zwischen diesem Konzeptraum und $CoRE_{CEPT}$ kanonisch und entsprechend eines allgemein akzeptierten Ansatzes [OMG CWM] integrieren (vgl. Abb. 22).
- Für die Konzepte von $CoRE_{CEPT}$ können Regeln zur automatischen Ableitung von *Repositories* für Entwurfsmodelle definiert werden. *Repositories* können genutzt werden, um konkrete Entwurfsmodelle, die auf *CoRE* basieren, zu hinterlegen, zu manipulieren bzw. in ihnen zu navigieren. *Repositories* sind die Grundlage für die Realisierung von $CoRE_{MAP}$ durch Entwicklungswerkzeuge.

Mit der Definition der Konzepte der Interaktionssicht geht diese Arbeit einen bedeutenden Schritt über die Modellierung ausschließlich struktureller Aspekte hinaus. Mit diesen Konzepten ist es möglich, für potentielle Bindungen zwischen *Instanzen* von CO-Typen Regeln anzugeben, die diese Bindungen erfüllen sollen. Die Angabe konkreter Regeln erfolgt nicht ausschließlich auf der Basis von Interfacetypen, sondern berücksichtigt die Erfassung von Umgebungskonstellationen. Die Modellierung der Regeltypen orientiert sich an [FK 98], erweitert diesen Ansatz jedoch um eine fallbasierte Erfassung der Regeln selbst. Diese Erweiterung ist ein zentraler Bestandteil der in [BKvH 00] vorgestellten Arbeiten zur Integration der automatischen Ableitung von Softwarekomponenten mit *Quality-of-Service*-Aspekte unterstützenden *Distributed-Processing*-Umgebungen.

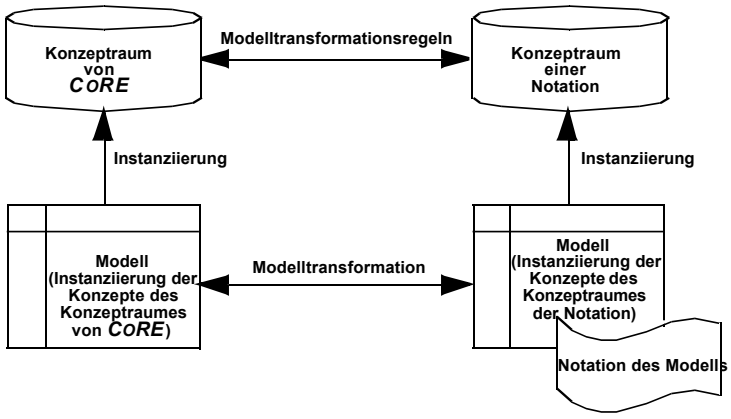


Abb. 22 Integration von $CoRE_{CEPT}$ mit dem Konzeptraum einer Notation

Neben den vorgestellten Konzepten zur Modellierung der Eigenschaften von potentiellen Bindungen zwischen COs wurde die Bereitstellung von Konstrukten zur Beschreibung des internen Verhaltens von Artefakten und damit der COs, die von diesen realisiert werden, untersucht. Einen geeigneten Ansatz können technologische Lösungen zum OMG-Prozeß „*Action Semantics for the UML Request for Proposals*" [OMG ASRFP] bilden, die derzeit innerhalb der OMG evaluiert werden. Eine mögliche derartige Lösung bildet die Adaption der Verhaltensbeschreibungsmittel von *Specification and Description Language* (SDL, [ITUTZ.100]). Die Integration von *CoRE* mit SDL wurde durch die Definition von Ableitungsregeln für die automatische Generierung von SDL-Spezifikationen aus Entwurfsmodellen erreicht.

KAPITEL 4

Resümee und Ausblick

In dieser Arbeit wird *CORE* (*Components Rapid Engineering*) als Integration von Entwicklungstechniken zur Unterstützung verschiedener Entwicklungsprozesse für verteilte Telekommunikationssoftwaresysteme definiert. Diese Entwicklungstechniken umfassen die objektorientierte Modellierung zur Unterstützung der Entwurfsphase, den Einsatz von Komponentenarchitekturen in der Integrations/Test- und Einsatz-/ Wartungsphase sowie die automatische Ableitung von Softwarekomponenten aus Entwurfsmodellen als Automatisierung des Übergangs zwischen Entwurfsphase und nachfolgenden Phasen - insbesondere der Implementierungsphase. Zur Definition von *CORE* wurden zunächst die Anforderungen analysiert, die die Domäne Telekommunikation impliziert. Als zentrale Anforderungen sind Offenheit dieser Softwaresysteme, Unterstützung von Gütebeschreibungen, flexible Adaptierbarkeit, Integrierbarkeit und Skalierbarkeit sowie Unterstützung von *Continuous-Media*-Interaktionen identifiziert worden. Neben diesen Anforderungen, die an zu entwickelnde Softwaresysteme selbst gestellt werden, konnten ebenfalls Anforderungen an die Entwicklungstechniken aufgestellt werden. Hierzu zählen insbesondere die konzeptionelle Offenheit sowie die umfassende Werkzeugunterstützung zur Realisierung kurzer Entwicklungszeiten.

Unter Berücksichtigung dieser Anforderungen wurden verschiedene verbreitete Entwicklungsprozesse analysiert. Es konnte gezeigt werden, daß sich die in *CORE* definierten Entwicklungstechniken in diese Prozesse integrieren lassen. Dabei wurde die zentrale Rolle der konzeptionellen Fundierung aller dieser Entwicklungstechniken gezeigt - die Konstruktion eines Konzeptraumes für *CORE* ist die Basis sowohl für die Definition begleitender Notationen als auch der Regeln zur automatischen Ableitung von Softwarekomponenten. Die Konstruktion dieses Konzeptraumes bezüglich seiner Kernkonzepte wurde ausgehend von einer Analyse existierender Ansätze und internationaler Standards (insbesondere *Reference Model for Open Distributed Processing* und CORBA *Component Model*) zunächst informal vorgenommen.

Die Kernkonzepte von $CORE_{CEPT}$ sind CO-Typ, Interfacetyp, Softwarekomponente und Artefakt. Der Zusammenhang zwischen funktionaler Dekomposition und konkreten Implementierungskonstrukten wird in *CORE* ebenfalls definiert. Damit ist es gelungen, für eine abstrakte, verteilungstransparente, funktionale Beschreibung eines Softwaresystems unter Adaption von RM-ODP-Konzepten die automatische Ableitung von konkreten Softwarekomponenten zu ermöglichen. Die Schlüsselkonzepte dazu sind Artefakt als

Abstraktion konkreter programmiersprachlicher Konstrukte in Entwurfsmodellen sowie die Integration der Abstraktion von Softwarekomponenten zur Definition des Ziels der automatischen Ableitung. Mit dem Konzept Interfacetyp ist es möglich, in Entwurfsmodellen einen gemeinsamen Kontext für Interaktionselemente unterschiedlicher Interaktionsarten zu definieren. Durch diese Konstruktion ist die Trennung von Interfacetypen nach Interaktionsarten - wie in RM-ODP gefordert - obsolet. Das Konzept CO-Typ erlaubt die Definition des Anbietens bzw. Benutzens von Interfacetypen unabhängig von ihren Interaktionsarten. Ebenso unabhängig von den durch einen Interfacetyp aggregierten Interaktionselementen lassen sich Konfigurationsaspekte von CO-Typen in Entwurfsmodellen erfassen. Mit weiteren Konzepten von $CoRE_{CEPT}$ wurde die Beschreibung der Güteanforderungen an Interaktionen zwischen Softwarekomponenten ermöglicht sowie die konzeptionelle Basis für die Anwendung von automatisierten *Deployment*-Ansätzen für Softwarekomponenten geschaffen.

Die Konstruktion von $CoRE_{CEPT}$ ermöglichte die Integration aller in *CoRE* definierten Entwicklungstechniken. $CoRE_{CEPT}$ sowie die darauf aufbauenden Entwicklungstechniken wurden durch die Autoren in Form von Software (Entwicklungswerkzeuge und Ausführungsumgebung für Softwarekomponenten) implementiert. Damit ist nachgewiesen, daß die prinzipielle Herangehensweise der Integration der Entwicklungstechniken basierend auf $CoRE_{CEPT}$ realisierbar ist. Die gesamte Werkzeugkette und die Ausführungsumgebung wurden mehrfach erfolgreich demonstriert, u.a. auf Tagungen von OMG und ITU-T. Zu diesen Anlässen konnten die Autoren während verschiedener Diskussionen mit Experten unterschiedlicher Domänen erkennen, daß das Anwendungspotential von *CoRE* nicht auf die Entwicklung von Softwaresystemen in der Telekommunikationsdomäne beschränkt ist.

CoRE wurde während der Entwicklung in von der Industrie beauftragten Forschungs- und Entwicklungsprojekten erfolgreich eingesetzt. Dadurch konnte bereits in frühen Entwicklungsstadien die praktische Einsetzbarkeit und Relevanz nachgewiesen werden. So wurden Softwarekomponenten, die mit Hilfe von *CoRE* aus Entwurfsmodellen erzeugt wurden, im Rahmen des EURESCOM-Projektes "*Distribution and Configuration Support for Distributed PNO Applications*" [EUP924] für die Projektdemonstratoren verwendet. Des weiteren wird die Entwicklung von *CoRE* im Rahmen des durch TINA-C initiierten TINA-*Fellowship*-Programms unterstützt [ComPoTel 00].

Während der Entwicklung von *CoRE* sind durch die Autoren verschiedene Aspekte einer möglichen Weiterentwicklung identifiziert worden. Auf der Basis dieser Identifikation wurden durch die Autoren bereits Forschungs- und Entwicklungsprojekte definiert, die sich gegenwärtig in der Startphase befinden [MDTS 01][ComPoTel00]. Untersuchungsgegenstände dieser Projekte sind:

- Erweiterung von $CoRE_{MAP}$ und $CoRE_{WARE}$ um alternative Zielsprachen und Komponentenarchitekturen,
- Erweiterung von $CoRE_{CEPT}$, $CoRE_{TATIONS}$ und $CoRE_{MAP}$ bezüglich der Unterstützung von *Business-Logic*-Beschreibungen in Entwurfsmodellen,
- Integration der automatischen Ableitung von Softwarekomponenten für Testsysteme aus Entwurfsmodellen in $CoRE_{MAP}$ und
- Ausbau der Konzepte von $CoRE_{CEPT}$ zur Gütebeschreibung und Garantie sowie dementsprechende Erweiterung von $CoRE_{TATIONS}$, $CoRE_{MAP}$ und $CoRE_{WARE}$.

Die in $CoRE_{MAP}$ integrierten Regeln zur automatischen Ableitung von Softwarekomponenten erzeugen neben der programmiersprachenunabhängigen Definition der Interfaces von Softwarekomponenten auch Implementierungsgerüste der Programmiersprache C++ [ANSIC++]. Eine Stärke von *CoRE* besteht in der Programmiersprachenunabhängigkeit von $CoRE_{CEPT}$. $CoRE_{MAP}$ kann ohne Modifikation von $CoRE_{CEPT}$ um Ableitungsregeln für alternative Programmiersprachen und Komponentenarchitekturen erweitert werden. Als Kandidaten sind hier die Komponentenarchitekturen *Enterprise Java Beans* [Sun EJB] und *CORBA Components* [OMGCCMI] sowie die Programmiersprache Java [Sun JAVA] zu nennen. Die diesbezüglichen Erweiterungen von $CoRE_{MAP}$ und $CoRE_{WARE}$ sind im Rahmen von [MDTS 01] und [ComPoTel 00]

geplant. Darüber hinaus können die Komponentenarchitektur *.net* [MS .net] und die Programmiersprache C# [MS C#] Relevanz erlangen, wenn die Konsolidierung dieser Technologien abgeschlossen ist.

In $CoRE_{CEPT}$ wird die konkrete Realisierung der Implementierungselemente durch Artefakte nicht im Modell erfaßt, muß also in den programmiersprachlichen Konstrukten, die aus der Anwendung von $CoRE_{MAP}$ resultieren, vorgenommen werden. Falls sich eine - bisher nicht existierende - technologische Lösung des im Dokument „*Action Semantics for the UML Request For Proposal*" [OMG AS RFP] vorgestellten Ansatzes einer plattform- und programmiersprachenunabhängigen Repräsentation programmiersprachlicher Mittel in Modellen in der Praxis bewährt, so kann genau dieser Ansatz zur Modellierung der Realisierung von Implementierungselementen durch Artefakte in $CoRE_{CEPT}$ integriert werden. Durch die objektorientierte Konstruktion von $CoRE_{CEPT}$ lassen sich die dann notwendigen Erweiterungen kanonisch integrieren. Voraussetzung für die Überführung der dann möglichen Beschreibungen von *Business Logic* in Entwurfsmodellen in Codemodule von Softwarekomponenten ist die Definition geeigneter Ableitungsregeln in Programmiersprachen. Wenn mit diesem Ansatz die Definition des Verhaltens von CO-Typen an deren unterstützten Interfaces möglich ist, so können neben der Ableitung von Codemodulen aus diesen Beschreibungen auch Softwarekomponenten automatisch gewonnen werden, die dieses Verhalten testen (vgl. [BHS+ 99]). Diesbezügliche Untersuchungen sind zentraler Bestandteil von [MDTS01].

Durch die Firma *KPN Research* wird die Komponentenarchitektur *Quality Aware Middleware (QuAM)* entwikkelt, die ein generisches Rahmenwerk zur Integration verschiedener Mechanismen zur Gütebeschreibung und Garantie realisiert [vHalt00]. Im Rahmen einer Forschungskooperation wurde bereits eine Konzeption entwickelt, deren Gegenstand eine Verbindung zwischen *CoRE* und *QuAM* ist [BKvH00]. Eine Präzisierung dieser Integration und die Realisierung der notwendigen Erweiterungen von *CoRE* wird angestrebt.

Der Verbreitungsgrad der Entwicklungstechniken von *CoRE* hängt in entscheidendem Maße von deren internationaler Standardisierung ab. Der modellzentrierten Herangehensweise von *CoRE* folgend umfaßt die Standardisierung dabei insbesondere den durch $CoRE_{CEPT}$ definierten Konzeptraum sowie die im Kontext von $CoRE_{MAP}$ festgelegten Ableitungsregeln zur automatischen Ableitung von Softwarekomponenten. Als Foren dieser notwendigen Standardisierungsaktivitäten bieten sich *Object Management Group* sowie ITU-T an, da sie sich mit der Entwicklung verteilter Softwaresysteme im Telekommunikationskontext beschäftigen.

Im Rahmen von *Object Management Group* wird seit November 2000 die Definition der Gesamtarchitektur *Model Driven Architecture* [OMG MDA] vorgenommen. Die Arbeiten an *CoRE* lassen sich in diesen Kontext einbetten. Die Autoren sind hier insbesondere innerhalb der Gruppen *CORBA Components Finalization Task Force* und *Implementers Group* aktiv. Es kann davon ausgegangen werden, daß Teile des in $CoRE_{CEPT}$ definierten Konzeptraumes sowie die diesem Teil zuzuordnende Menge von Ableitungsregeln für Softwarekomponenten von $CoRE_{MAP}$ Bestandteil der finalen Spezifikation des CORBA-Komponentenmodells werden. Darüber hinaus können in neu zu definierenden Standardisierungsprozessen für Erweiterungen des CORBA-Komponentenmodells weitere Aspekte von *CoRE* einfließen [OMG DeplRFP].

Im Kontext von *ITU-T Study Group 10 Question 2/11* wird eine neue Version der Beschreibungssprache ITU-ODL sowie die Beschreibungssprache *Distribution and Configuration Language* (DCL) definiert. Hier werden sowohl die durch *CoRE* definierte Herangehensweise als auch die in *CoRE* realisierten Entwicklungstechniken eingebracht. Im Rahmen dieses Standardisierungsprozesses betrachten die Autoren den durch *CoRE* definierten Konzeptraum und die Ableitungsregeln für Softwarekomponenten für die Programmiersprache C++ als wesentlichen Beitrag. Erstmals wird dabei im Rahmen von ITU-T nicht die konkrete Syntax einer Beschreibungssprache sondern deren semantische Fundierung in Form eines formalisierten Konzeptraumes in den Mittelpunkt gestellt.

In den folgenden Bänden werden die Entwicklungstechniken von *CoRE* präzise definiert und ihre Anwendung anhand von Beispielen illustriert. $CoRE_{CEPT}$ und $CoRE_{TATIONS}$ sind Untersuchungsgegenstand von [CoRE II], $CoRE_{MAP}$ und $CoRE_{WARE}$ werden in [CoRE III] diskutiert.

REFERENZEN

[ANSIC++] ISO/IEC 14882:1998: "*Programming languages - C++*", ISO '98

[BF 01] Born, Fischer: "*Object Definition Language*" Telektronikk 04/2001, Kjeller, Norway '01

[BFL+ 99] Born, Fischer, Löwis, Krüger, Ulbricht: "*Service Composition in a TINA Enviroment*", Proceedings of the TINA 1999 Conference, Oahu, USA '99

[BH 98] Born, Hoffmann: "*An Object-Oriented Design Methodology for Distributed Systems*", Proceedings of the Technology of Object-Oriented Languages and Systems 1998 (TOOLS Pacific 1998) Conference, Melbourne, Australia '98

[BH 98a] Born, Hoffmann: "*Advanced Distribution and Configuration Support for Distributed Applications*" Proceedings of the Workshop on Distributed Object Technologies for Telecoms Networks (DOT'98), Heidelberg, Germany '98

[BHK 01] Born, Holz, Kath: "*Manufacturing Software Components from Object-Oriented Design Models*", Proceedings of the 5th International Enterprise Distributed Object Computing Conference (EDOC 2001), Seattle, USA '01

[BHL+ 98] Born, Hoffmann, Li, Schieferdecker: "*Applying a Framework Approach with Validation to the Design of Telecommunication Services*": Proceedings of the International Conference on Communication Technologies (ICCT'98), Beijing, China '98

[BHL+ 99] Born, Hoffmann, Li, Schieferdecker: "*Using Formal Methods for the Design of Telecommunication Services*" Proceedings of the Conference on Formal Methods for Object Oriented Distributed Systems (FMOODS) '99, Florence, Italy '99

[BHS+99] Born, Hoffmann, Schieferdecker, Vassiliou-Gioles, Winkler: "*Performance Testing of a TINA Platform*" : Proceedings of the TINA 1999 Conference, Oahu, USA '99

[BHW98] Born, Hoffmann, Winkler: "*SDL Enhancements and Integration into the Design-Lifecycle of Telecommunication Services*", Proceedings of the International Conference on Communication Technologies (ICCT'98), Beijing, China '98

[BHW 98a] Born, Hoffmann, Winkler: "*The ITU-ODL to C++ Mapping and its Integration into an SDL based Design-Methodology for Telecommunication Services*", Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC (SAM´98), Berlin, Germany '98

[Bitk 00] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien: "*Informationstechnik und Telekommunikation im Dauerhoch*", Bitkom-Presseinformation, http://www.bitkom.org/presse/pr230200.htm, BitKom '00

[Bitk 00a] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien: "*Wege in die Informationsgesellschaft - Status quo und Perspektiven Deutschlands im internationalen Vergleich*", BitKom '00

[BK 00] Born, Kath: "*From TINA-ODL Towards a Component Oriented Design Method*". Proceedings of the TINA 2000 Conference, Paris, France '00

[BK 00a] Born, Kath: "*Code Generation for Component based Telecommunication Service Development*", Proceedings of the SoftCom 2000 Conference, Split, Croatia '00

[BK 00b] Born, Kath: "*Customizing UML for Component Design*", Proceedings of the 1st OMG Workshop: UML In The .com Enterprise - Modeling CORBA, Components, XML/XMI And Metadata, Palm Springs, USA '00

[BK 01] Born, Kath: "*Distributed Applications: From Models to Components*", EURESCOM Workshop on Middleware in Telecommunications, Kjeller, Norway '01

[BK 01a] Born, Kath: "*The DOT Profile*", http://www.dot-profile.de/

[BK 01b] Born, Kath: "*Distributed Applications: From Models to Components*", Presentation at the Object Management Group TC Meeting, document number telecom/01-03-02, Irvine, USA '01

[BKH 00] Born, Kath, Holz: "*A UML Profile for Integrated Design and Development of Distributed Applications*", Proceedings of the Technology of Object-Oriented Languages and Systems 2000 (TOOLS Pacific 2000) Conference, Sydney, Australia '00

[BKvH00] Born, Kath, v. Halteren: „*Modeling and Runtime Support for Quality of Service in Distributed Component Platforms*", Work in Progress Paper at 11th Annual IFIP/IEEE International Workshop on "Distributed Systems: Operations and Management", Austin, USA '00

[Box99] Box: "*COM - The Component Object Model*", Addison-Wesley '98

[BMBF 00] Bundesministerium für Bildung und Forschung: "*Analyse und Evaluation der Softwareentwicklung in Deutschland*", http://www.dlr.de/IT/IV/Studien/evasoft_abschlussbericht.pdf, BMBF '00

[BS98] Born, Strick: "*Development Tools for Distributed Services*", European Research Consortium for Informatics and Mathematics News No.34, Sophia Antipolis, France '98

[BSH 01] Born, Schieferdecker, Hoffmann: "*The Deployment and Configuration Language*", EURESCOM Workshop on Middleware in Telecommunications, Kjeller, Norway '01

[BSL 00] Born, Schieferdecker, Li: "*Test Framework for Component-Based Systems*", 20th International Conference on Distributed Computing Systems (ICDCS' 2000) with International Workshop on Distributed System Validation and Verification (DSVV'2000), Taipei, Taiwan '00

[BSL 00a] Born, Schieferdecker, Li: "*UML Framework for automated Generation of component based test systems*", 1st International Conference on Software Engineering Applied to Networking & Parallel/ Distributed Computing (SNPD'00), Reims, France '00

[CCMP00] Research Project Description: "*Implementation of CORBA Component Model Session Container*", Berlin, Germany '00

[ComPoTel 00] Research Project Description: "*Component Oriented Design in Telecommunications - Concepts, Notation and Middleware Platform Support*", TINA Fellowship Program, Red Bank, USA '00

[*CoRE* I] Born, Kath: "*CoRE - Komponentenorientierte Entwicklung offener, verteilter Softwaresysteme im Telekommunikationskontext, Band I - Entwicklungsprozesse und Entwicklungstechniken*"

[*CoRE* II] Born: "*CoRE - Komponentenorientierte Entwicklung offener, verteilter Softwaresysteme im Telekommunikationskontext, Band II: Konzeptraum und Notationen*"

[*CoRE* III] Kath: "*CoRE - Komponentenorientierte Entwicklung offener, verteilter Softwaresysteme im Telekommunikationskontext, Band III: Plattformunterstützung und Ableitungsregeln für Softwarekomponenten*"

[DBAG98] "*AEROSPACE*", Magazine of Daimler-Benz Aerospace AG, 1/98, Stuttgart, Germany '98

[DBB+ 01] Dubois, Born, Boehme, Fischer, Holz, Kath, Neubauer, Stoinski: "*Distributed Systems: From Models to Components*", Proceedings of the 10th SDL Forum Conference, Copenhagen, Denmark '01

[DSTCdMOF] Distributed Systems Technology Centre: "*dMOF - An OMG Meta Object Facility Implementation*", http://www.dstc.edu.au/products/CORBA/MOF

[EUP715] EURESCOM Project P715 Deliverable: "*Deliverable 2: Experiments on the EURESCOM Services Platform, Final Report*", Heidelberg, Germany '99

[EUP910] EURESCOM Project P910 Deliverable: "*Deliverable 7: Report on Telecommunication Application Domains*", Heidelberg, Germany '01

[EUP924] EURESCOM Project P924: "*Deployment and configuration support for distributed PNO applications*", http://www.eurescom.de/public/projects/P900-series/p924/P924

[EUP924a] EURESCOM Project P924 Deliverable: "*Deliverable 2: Notation and semantics for deployment and initial configuration*", Heidelberg, Germany '01

[FBH+ 99] Fischbeck, Born, Hoffmann, Winkler, Baudis, Böhme, Fischer: "*SDL enhancements and application for the design of distributed services*": Proceedings of the 9th SDL Forum Conference, Montreal, Canada '99

[FFB 98] Fischer, Fischbeck, Born: "*SDL und ODL im Entwicklungsprozess von Telekommunikationssystemen*": König, Langendörfer (eds.), Formale Beschreibungen für Verteilte Systeme, Shaker Verlag, Aachen, Germany '99

[FFH+ 97] Fischbeck, Fischer, Holz, Kath, v. Löwis, Schröder: "*Improving the Development and Validation of Viewpoint Specifications*", Proceedings of the Conference on Formal Methods for Object Oriented Distributed Systems (FMOODS) '97, Centerburry, UK '97

[FHK+ 98] Fischbeck, Holz, Kath, Vogel: "*Flexible Support of ORB Interoperability*", Proceedings of the Interworking '98 Conference, Ottawa, Canada '98

[FK 98] Frolund, Koistinen: "*QML: A Language for Quality of Service Specification*", White Paper, Hewlett-Packard Laboratories '98

[FK 99] Fischbeck, Kath: "*CORBA Interworking over SS.7*", Proceedings of the Conference on Intelligence in Services in Networks, Barcelona, Spain '99

[FKB 00] Fischer, Kath, Born: "*TINA-ODL and Component Based Design*", Proceedings of the 6th International Conference on Object-Oriented Information Systems 2000, London, UK '00

[FKH+ 98] Fischbeck, Kath, Holz, Geipl, Vogel: "*Operational and Stream Interactions in a B-ISDN based Environment*", Proceedings of the INFORMS 1998 Conference, Boca Raton, USA '98

[FTK+00] Funabashi, Toyouchi, Kanai, Uchihashi, Kobayashi, Hakomori, Yoshida, Strick, Born: "*Development of Open Service Collaborative Platform for Coming ECs by International Joint Efforts*" Procceedings of International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet, Rom, Italy '00

[Gart99] Gartner Executive Report: "*Application Integration: Putting the Pieces of the Puzzle Together*" Gartner Group '99

[Gart99a] Gartner Strategic Analysis Report: "*Middleware Deployment Trends: Survey of Real-World Enterprise Applications*" Gartner Group '99

[GHJ+ 99] Gamma, Helm, Johnson, Vlissides: "*Elements of Reusable Object-Oriented Software*", Addison-Wesley '99

[HKG+ 97] Holz, Kath, Geipl, Lin, Vogel: "*The CAMOUFLAGE Project -Introduction Of TINA Into Telecommunication Legacy Systems*", Proceedings of the TINA 1997 Conference, Santiago de Chile, Chile '97

[HWB 97] Hoffmann, Winkler, Born, Fischer, Fischbeck: *"Towards a Behavioural Description of ODL"* Proceeding of the TINA 1997 Conference, Santiago, Chile '97

[HV 99] Henning, Vinoski: "*Advanced CORBA Programming with C++*", Addison-Wesley '99

[InstallShield] InstallShield Software Corporation: "InstallShield 6.3", http://www.installshield.com/

[ITUTX.292] ITU-T Recommendation X.292: *"OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – The Tree and Tabular Combined Notation (TTCN)"*, ITU '98

[ITUTX.608] ITU-T Recommendation X.608: "*Abstract Syntax Notation One*", ITU-T '99

[ITUTX.902] ITU-T Recommendation X.902 |ISO/IEC 10746-2: *"Open Distributed Processing - Reference Model Part 2"*, ITU-T/ISO '95

[ITUTX.903] ITU-T Recommendation X.903 | ISO/IEC 10746-3: *"Open Distributed Processing - Reference Model Part 3"*, ITU-T/ISO '95

[ITUTX.904] ITU-T Recommendation X.904 | ISO/IEC 10746-4: *"Open Distributed Processing - Reference Model Part 4"*, ITU-T/ISO '95

[ITUTZ.100] ITU-T Recommendation Z.100: "*Specification and Description Language*", ITU-T '00

[ITUTZ.109] ITU-T Recommendation Z.109: "*SDL combined with UML*", ITU-T '00

[ITUTZ.130] ITU-T Recommendation Z.130: "*The ITU-T Object Definition Language*", ITU-T '99

[Kath 99] Kath: "*How to get Behavioural Information on CORBA Systems?*" Proceedings of the Workshop on the Application of Distributed Object Technology, Heidelberg, Germany '99

[KHF+97] Kath, Holz, Fischer, Geipl, Vogel: "*Provision of TINA Object Interaction through B-ISDN in ATM networks*", Proceedings of the GLOBECOM Conference, Phoenix, USA '97

[KKS00] Kath, Kleinschmidt, Stoinski: *"CPE Architecture for TINA Services (CATS II): Final Project Deliverable"*, document number CATS-II HU 006, Tokio, Japan '00

[KS00] Kath, Stoinski: "*Project Proposal: Platform for Authoring and Composition of Interactive Content (PACIFIC)*", Projektvorbereitungsdokument, Berlin, Germany '00

[KST+ 01] Kath, Stoinski, Takita, Tsuchiya: "*Middleware Platform Support for Multimedia Content Provision*", Proceedings of the 3G Technologies and Applications Conference, Heidelberg, Germany '01

[KT 99] Kath, Takita: "*OMG A/V Streams and TINA NRA: An integrative Approach*", Proceedings of the TINA 1999 Conference, Oahu, USA '99

[KvHS+ 00] Kath, v. Halteren, Stoinski, Wegdam, Fisher: "*Middleware Platform Management Based on Portable Interceptors*", Proceedings of 11th Annual IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Austin, USA '00

[MDTS 01] Forschungs- und Entwicklungsprojekt: "*Model Driven Development of Distributed Telecommunication Systems*", Bundesministerium für Bildung und Forschung, Berlin, Germany '01

[MICO] Römer, Puder et. al.: "*MICO for C++*", http://www.mico.org/

[MS .net] Microsoft Corporation: "*The .NET Framework*", Micosoft Developer Network Online, http://msdn.microsoft.com/net/framework/default.asp

[MS C#] Microsoft Corporation: "*C# Introduction and Overview*", Micosoft Developer Network Online, http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp

[MS COM] Microsoft Corporation: "*The Component Object Model*", Microsoft Press '98

[Neu 95] Neumann: "*Objektorientierte Entwicklung von Software-Systemen*", Addison-Wesley '95

[OMGASRFP] Object Management Group: *"Action Semantics for the UML Request For Proposal"* OMG document ad/98-11-01

[OMGAVStreams] Object Management Group: "*Control and Management of Audio/Video Streams*", OMG document formal/00-01-03

[OMGCCM I] BEA Systems et. al.: *"CORBA Components - Volume I"*, OMG document orbos/99-07-01

[OMGCCM II] BEA Systems et. al.: "*CORBA Components - Volume II*", OMG document orbos/99-07-02

[OMGCCM III] BEA Systems et. al.: "*CORBA Components - Volume III*", OMG document orbos/99-07-03

[OMGCCM RFP] Object Management Group: "*CORBA Component Model RFP, Final Version*", OMG document orbos/97-06-12

[OMGCORBA] Object Management Group: „*The Common Object Request Broker: Architecture and Specification, Revision 2.4.2*", OMG document formal/01-02-33

[OMGCORBAES] Object Management Group: "*Event Service Specification, Version 1.1*", OMG document formal/01-03-01

[OMGCORBAIDL] Object Management Group: „CORBA 2.4.2 IDL Syntax and Semantics", OMG document formal/01-02-39

[OMGCORBANaS] Object Management Group: "*CORBA Naming Service*", OMG document formal/01-02-65

[OMGCORBANoS] Object Management Group: "*Notification Service Specification, Version 1.0*", OMG document formal/00-06-20

[OMGCORBAP] Data Access Corporation et. al.: "*UML Profile for CORBA*", OMG document ad/00-02-02

[OMGCWM] Object Management Group: "*Common Warehouse Metamodel (CWM) Specification, Version 1.0*", OMG document ad/01-02-01

[OMGC++Map] Object Management Group: "*C++ Language Mapping Specification*", OMG document formal/99-07-41

[OMGDeplRFP] Object Management Group: "*Deployment and Configuration of Distributed Applications Draft RFP*", OMG document orbos/01-04-12

[OMGMDA] Object Management Group: "*Model Driven Architecture*", OMG document omg/00-11-05

[OMGMICRFP] Object Management Group: "*Multiple Interfaces and Composition RFP*", OMG document orb/96-01-04

[OMGMOF1.3] Object Management Group: "*Meta Object Facility, Version 1.3*", OMG document formal/00-04-03

[OMGMOFP] Object Management Group: "*UML Profile for MOF*", OMG document ad/01-02-29

[OMGPI] BEA Systems et. al.: "*Portable Interceptors*", OMG document orbos/99-12-02

[OMGPSS2.0] Object Management Group: "*Persistent State Service 2.0*", OMG document orbos/99-07-07

[OMGReqUMLP] Object Management Group: "*Requirements for UML Profiles*", OMG document ad/99-12-32

[OMGSPEM 01] IBM, Rational Software, Fujitsu/DMR, SOFTEAM, Unisys, Nihon Unisys Ltd., Alcatel, Q-Labs: "*Software Process Engineering Management: The Software Process Engineering Metamodel (SPEM)*", OMG document ad/2001-03-08

[OMGSPERFP] Object Management Group: "*Software Process Engineering (SPE) Management Request for Proposal*", OMG document ad/99-11-04

[OMGUML1.3] Object Management Group: "*OMG Unified Modeling Language Specification, Version 1.3*", OMG document ad/99-06-08

[OMGXMI1.1] Object Management Group: "*XML Metadata Interchange (XMI) version 1.1*" OMG document formal/00-11-02

[OOC] Object Oriented Concepts Inc.: "*ORBacus™ for C++ and Java*", http://www.ooc.com/ob/

[OOCa] Object Oriented Concepts: "*Customer Success Stories*", http://www.ooc.com/customers/telecom.html

[OOCJTC] Object Oriented Concepts Inc.: "*Java™-like Threads for C++*", http://www.ooc.com/jtc/

[Pom 91] Pomberger: "*Prototyping-Oriented Software Development - Concepts and Tools*", Structured Programming, Vol. 12, Nr. 1 '91

[PS 94] Pagel, Six: "*Software Engineering - Band 1: Die Phasen der Softwareentwicklung*", Addison-Wesley '94

[Rational] Rational Software Corp.: "*Customer Stories*", http://programs.rational.com/success/Success_Result.cfm

[RationalRose] Rational Software Corp.: "*Rational Rose*", http://www.rational.com/products/rose/index.jsp/

[RationalRUP] Rational Software Corp.: "*Rational Unified Process: Best Practices for Software Development Teams*" Rational Software Corp. White Paper, http://www.rational.com/products/whitepapers/100420.jsp

[Ray95] Raymond: "*Reference Model of Open Distributed Processing (RM-ODP): Introduction*" Proceedings of International Conference of Open Distributed Processing, Brisbane, Australia '95

[RFCMIME] Internet RFC 2046: "*Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*", IETF '98

[Roy 70] Royce: "*Managing the Development of Large Scale Software Systems*", Proceedings of WESCON Conference, Los Angeles, USA '70

[SB97] Strick, Born: "*Developing TINA Services with ODL and SDL*" Proceedings of the Workshop on Distributed Object Technologies for Telecoms Networks (DOT '97), Heidelberg, Germany '97

[Sun 01] Sun Microsystems White Paper: „*Network Application Strategies for Telecom: Introduction Component Architectures for Enhanced Services*", http://www.sun.com/software/solutions/third-party/software/whitepapers/Java.Telco.pdf

[Sun EJB] Sun Microsystems: "*Enterprise JavaBeans TM*" http://www.sun.com

[Sun EJBa] Sun Microsystems: "*Industry Opinions On Enterprise JavaBeansTM (EJBTM) vs. COM+/MTS*", http://java.sun.com/products/ejb/ejbvscom.html

[Sun JAVA] Sun Microsystems: "*Java 2 Platform Standard Edition*", http://www.javasoft.com/j2s

[Sun SSL] Sun Microsystems, Netscape Corp.: "*Introduction to SSL*", http://docs.iplanet.com/docs/manuals/security/sslin/index.htm

[Szy99] Szyperski "*Component Software - Beyond Object-Oriented Programming*", Addison-Wesley '99

[TINA] TINA-C: "http://www.tinac.com"

[TINACMC] TINA-C: "*Computational Modeling Concepts*",TINA-C '98

[TINAReq] TINA-C: "*Requirements upon TINA-C Architecture*", TINA-C '95

[TINASA] TINA-C: "*TINA Service Architecture 5.1*", TINA-C '97

[TTK+ 00] Tsuchiya, Takita, Kath, Stoinski: "*Authoring, Composition, and Delivery of Interactive Contents by the use of Multimedia Contents Mill*", Proceedings of the TINA 2000 Conference, Paris, France '00

[TTK+ 00a] Tsuchiya, Takita, Kath, Stoinski: "*'Multimedia Contents Mill' – A Platform for Authoring and Delivery of Interactive Multimedia Contents*", Proceedings of the SoftCom 2000 Conference, Split, Croatia '00

[UnisysUREP] Unisys Corp.: "*Universal Repository*", http://www.unisys.com/marketplace/urep/

[vHalt00] A.T. van Halteren: "*A reflective QoS provisioning service for object middleware*", Workshop on Reflective Middleware (RM 2000), co-located with the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '00), New York, USA '00.

[VSB+ 99] Vassiliou-Gioles, Schieferdecker, Born, Winkler, Li: "*Configuration and Execution Support for Distributed Tests*" Proceedings of the 12th International Workshop on Testing of Communicating System (IWTCS 99), Budapest, Hungary '99

[Web01] Weber: "*Quo Vadis Software Engineering*", Gesellschaft für Informatik, Softwaretechnik-Trends, Band 21 Heft 1, http://pi.informatik.uni-siegen.de/stt/21_1/index.html, GI '00

[WS 96] Weck, Szyperski: „*Do We Need Inheritance?*" Proceedings of the Workshop on Composability Issues in Object-Orientation (at ECOOP'96), Linz, Austria '96

[WWF+ 95] Wasowski, Witaszek, Fischer, Holz, Lau, Kath: "*Co-Simulation of Hybrid SDL and VHDL Specifications*", Proceedings of the 9th ESM Conference, Prague, Chzech Republic '95

[W3CHTTPNG] Janssen et. al.: „*HTTP-ng Architectural Model*", W3C Internet Draft, www.w3c.org

[W3COSD] Marimba Incorporated, Microsoft Corporation: "*The Open Software Description Format (OSD)*", submitted to W3C, http://www.w3.org/TR/NOTE-OSD.html, W3C '97

[W3CSMIL] W3C Recommendation: "*Synchronized Multimedia Integration Language (SMIL 2.0) Specification* ", W3C Proposed Recommendation, http://www.w3.org/TR/2001/PR-smil20-20010605/, W3C '01

[W3CXML] W3C Recommendation: "*Extensible Markup Language (XML) 1.0 (Second Edition)*", http://www.w3.org/TR/2000/REC-xml-20001006, W3C '00

[W3CXMLDT] W3C Recommendation: "*XML Schema Part 2: Datatypes*", http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/, W3C '01

ABKÜRZUNGEN

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CCM	CORBA Component Model
CDR	Common Data Representation
CIDL	Component Implementation Definition Language
CIF	Component Implementation Framework
CO	Computational Object
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CoRE	Components Rapid Engineering
DCOM	Distributed COM
DTD	Document Type Definition
EJB	Enterprise Java Beans
HTTP-NG	HTTP Next Generation
IDL	Interface Definition Language

IIOP	Internet Inter-ORB Protocol
OCL	Object Constraint Language
ODL	Object Definition Language
OMG	Object Management Group
OSD	Open Software Description
MDA	Model Driven Architecture
MOF	Meta Object Facility
MP3	MPEG Audio Layer-3
MPEG	Moving Picture Experts Group
RFP	Request for Proposals
RMI	Remote Methode Invocation
RM-ODP	Reference Model for Open Distributed Processing
RUP	Rational Unified Process
SDL	Specification and Description Language
SPEM	Software Process Engineering Metamodel
TINA	Telecommunications Information Networking Architecture
UDP	User Datagram Protocol
UML	Unified Modeling Language
UUID	Universal Unique Identifier
XMI	XML Metadata Interchange
XML	Extensible Markup Language

INDEX

ANHANG A *Eingeführte Termini*

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Artefakt	*artifact*	Abstraktion konkreter programmiersprachlicher, in Softwarekomponenten in Form von Codemodulen enthaltener Konstrukte (z.B. Klassen im Falle von objektorientierten Programmiersprachen) Artefaktinstanzen realisieren das Verhalten, den Zustand und die Identität von COs	CCM
Assemblage	*assembly*	Beschreibung eines verteilten Softwaresystems durch Angabe der beteiligten CO-Typen und deren initialer Konfiguration	CCM
Attribut (eines Interfacetyps)	*attribute*	Spezielle Variante von Operationen im Sinne einer verkürzten Schreibweise für *get*- und *set*-Operationen für einen bestimmten Datentyp	CORBA
Ausführungszeit	*runtime*	Zeit, in der die in Softwarekomponenten enthaltenen Codemodule durch eine Maschine ausgeführt werden	
Ausnahme	*exception*	Spezielle Art der Operationsterminierung im Falle von Fehlern	RM-ODP

Tab.2 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Bindung	*binding*	Konzept zur Beschreibung des Austausches von Referenzen auf Instanzen der zu *Port*-Definitionen gehöhrenden Interfacetypen unter Beachtung der Art der *Port*-Definition (*Used*- oder *Provided- Port*-Definition)	RM-ODP, CCM
Bindungsfall	*binding case*	Zusammenfassung von COs, Bindungen dieser COs und zu diesen Bindungen gehörigen Bindungsregeln	
Bindungsregel	*binding rule*	Beschreibung von geforderten Eigenschaften von Interaktionen, die auf der Grundlage von Bindungen entstehenden, formuliert als Regeln über Kontrakttypen	RM-ODP
Component-Support- Plattform	*component support platform*	Plattform, die *Deployment*- und Ausführungsunterstützung für Softwarekomponenten unter Benutzung einer *Distributed-Processing*-Umgebung bereitstellt	
Computational- Objekt	*computational object*	Objekt, das durch funktionale Dekomposition eines Softwaresystems während dessen Modellierung entsteht	RM-ODP, TINA
Computational- Objekttyp	*computational object type*	Beschreibung von *Computational*-Objekten, die zu deren Instanziierung benutzt wird	RM-ODP, TINA
Consume-Definition	*consume*	Interaktionselement der Signalinteraktion, beschreibt durch Angabe eines Signaltyps die Möglichkeit des Konsumierens eines Signals dieses Signaltyps im Kontext eines Interfaces	CCM
Continuous-Media-, Signal- und operationale Interaktion	*continous media, signal and operational Interaction*	Interaktion zwischen COs unter Verwendung von operationalen, Signal- oder *Continous-Media*-Interaktionselementen (Operation, Attribut, *Consume*, *Produce*, *Sink*, *Source*)	RM-ODP, TINA (operationale und *Continous-Media*-Interaktion)
Datentyp	*data type*	Element eines Datentypsystems (z.B. CORBA-IDL) und Basis von Informationsmodellen, dient der strukturierten Repräsentation von Information	CORBA
Distributed-Processing- Umgebung	*distributed processing environment*	Technologische Grundlage zur Unterstützung der Interaktion von Objekten eines verteilten Softwaresystems	TINA
Implementierungs-element	*implementation element*	Beschreibung einer Relation zwischen Interaktionselement und Artefakt mit der Bedeutung, daß eine Instanz des Artefakts für die Realisierung des Verhaltens des Interaktionselements verantwortlich ist.	
Implements-Relation	*implements*	Relation zwischen Artefakten und CO-Typen mit der Bedeutung, daß Instanzen der Artefakte das Verhalten der COs dieses CO-Typs realisieren	

Tab.2 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
initiales CO	*initial CO*	Instanz eines CO-Typs, die zu Beginn der Ausführungszeit des Softwaresystems erzeugt wird	CCM (nicht als CO sondern als CORBA *component* bezeichnet)
initiale Konfiguration	*initial configuration*	Beschreibung der initialen COs und deren initialen Bindungen	CCM
initiale Bindung	*initial binding*	Bindung, die zu Beginn der Ausführungszeit eines Softwaresystems erzeugt wird	CCM
Instanziierungs-muster	*instantiation policy*	Beschreibung, nach welchen Regeln zur Ausführungszeit Instanzen des durch ein Artefakt modellierten programmiersprachlichen Konstruktes angelegt werden sollen	
Interaktionselement	*interaction element*	Generalisierung der Konzepte Operation, Attribut, *Sink*, *Source*, *Consume*, *Produce*	
Interface	*interface*	auf der Grundlage eines Interfacetyps im Kontext eines COs entstehende referenzierbare Zusammenfassung von potentiellen Interaktionen eines COs	RM-ODP
Interfacetyp	*interface type*	Aggregation von Interaktionselementen als benannter, identifizierbarer Endpunkt von potentieller Interaktion	RM-ODP
Kontrakttyp	*contract type*	Datentyp, der Elemente zur Definition von Eigenschaften der aufgrund von Bindungen zustandekommenden Interaktionen beinhaltet	QML
Konzeptraum	*concept space*	Die Gesamtheit aller für die objektorientierte Modellierung in einem Anwendungsgebiet verwendbaren Konzepte und deren Beziehungen	
Maschine	*node*	Gerät, das zur Ausführung der in Softwarekomponenten enthaltenen Codemodule geeignet ist (i.allg. Computer)	RM-ODP
Medienmenge	*media set*	Aggregation von Medien	
Medientyp	*media type*	Vorschrift für die Codierung, die Übertragung und Decodierung der Mediendaten eines Mediums	
Medium	*medium*	Abstraktion einer multimedialen Information	
Metamodell	*metamodel*	Modell, daß das Instrumentarium zur Definition von Modellen beschreibt	MOF
Modellklasse	*viewpoint*	Definition einer Einschränkung auf bestimmte Aspekte von Entwurfsmodellen durch Angabe von hierfür relevanten Konzepten und Beziehungen eines Konzeptraumes	RM-ODP

Tab.2 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Multiple-Port-Definition	*multiple port*	*Port*-Definition, auf deren Grundlage eine Vielzahl von Interfacereferenzen zur Ausführungszeit eines COs hinterlegbar oder beschaffbar ist	CCM (dort nur für benutzte Interfacetypen)
Namensraum	*namespace*	Konzept zur Strukturierung von Namen der Elemente eines Modells	RM-ODP
Objekt	*object*	Modell einer Entität von Interesse in der betrachteten Anwendungsdomäne	RM-ODP
Objektumgebung	*environment of an object*	Teil des Softwaresystems, der nicht Bestandteil des Objektes ist	RM-ODP
Operation	*operation*	Interaktionselement der operationalen Interaktion, beschrieben durch Parameter und mögliche Arten der Terminierung	RM-ODP, CORBA
Parameter	*parameter*	Identifizierbarer Bestandteil einer Operation, definiert die Richtung des Informationsflusses bei der Interaktion und einen der Information zugrundeliegenden Datentyp	CORBA
Port-Definition	*port*	Beschreibung der Möglichkeit der Hinterlegung bzw. Beschaffung von Referenzen auf Interfaces eines entsprechenden COs zur Ausführungszeit	CCM
Prädikat	*predicate*	Abstraktion von zu wahr oder falsch evaluierbaren Ausdrücken	RM-ODP
Produce-Definition	*produce*	Interaktionselement der Signalinteraktion, beschreibt durch Angabe eines Signaltyps die Möglichkeit des Versendens eines entsprechenden Signals im Kontext eines Interfaces	CCM
Provided-Port-Definition	*provides*	auf einer *supports*-Relation basierende *Port*-Definition zur Beschreibung der Möglichkeit der Beschaffung von Referenzen auf Interfaces eines COs zur Ausführungszeit	CCM
Realize-Relation	*realize*	Zuordnung von CO-Typen zu Softwarekomponenten	UML
Requires-Relation	*requires*	Relation zwischen Interfacetyp und CO-Typ mit der Bedeutung, daß COs dieses CO-Typs Interfaces dieses Interfacetyps von ihrer Umgebung erwarten	TINA
Sicht	*viewpoint*	s. Modellklasse	RM-ODP
Signal	*signal*	Atomare Nachricht, die asynchron und entkoppelt zwischen COs ausgetauscht wird, entsteht auf der Basis eines Signaltyps	RM-ODP

Tab.2 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Signalparameter	*signal parameter*	Identifizierbare Angabe eines Datentyps im Kontext eines Signaltyps	RM-ODP
Signaltyp	*signaltype*	Beschreibung von Signalen, die zu deren Instanziierung im Kontext von Signalinteraktion verwendet wird, Aggregation von Signalparametern	RM-ODP
Single-Port-Definition	*single port*	*Port*-Definition, auf deren Grundlage eine einzelne Interfacereferenz zur Ausführungszeit eines COs hinterlegbar oder beschaffbar ist	CCM
Sink-Definition	*sink*	Interaktionselement der *Continous-Media*-Interaktion, beschreibt durch Angabe einer Medienmenge die Möglichkeit des Empfangs der Medien auf der Basis eines geeigneten Medientyps im Kontext eines Interfaces	TINA
Software-komponente (modelliert)	*software component*	Abstraktion einer Softwarekomponente im Modell (Abstraktion von den Instruktionssequenzen)	[Szy99]
Software-komponente (real)	*software component*	physikalisch repräsentierte Entität, bestehend aus in Codemodulen zusammengefaßten Instruktionssequenzen, die Ausführung einer Softwarekomponente führt zur Inkarnation von Objekten	[Szy99]
Softwarepaket	*software package*	Zusammenfassung von Softwarekomponenten eines realen Softwaresystems	CCM, OSD
Softwaresystem (real)	*software system*	System bestehend aus Softwarekomponenten	[Szy99]
Source-Definition	*source*	Interaktionselement der *Continous-Media*-Interaktion, beschreibt durch Angabe einer Medienmenge die Möglichkeit des Sendens der Medien auf der Basis eines geeigneten Medientyps im Kontext eines Interfaces	TINA
supports-Relation	*supports*	Relation zwischen Interfacetyp und CO-Typ mit der Bedeutung, daß COs dieses CO-Typs Interfaces dieses Interfacetyps ihrer Umgebung bereitstellt	TINA
Terminierung	*termination*	Ende einer gerufenen Operation	RM-ODP
Used-Port-Definition	*uses*	auf einer *requires*-Relation basierende *Port*-Definition	CCM
Zustandsattribut	*state attribute*	spezieller Datentyp zur Repräsentation von Zustandsinformationen von CO-Typen	RM-ODP, CCM

Tab.2 Verwendete Termini in der Übersicht

CoRE - Komponentenorientierte Entwicklung offener verteilter Softwaresysteme im Telekommunikationskontext

Band II - Konzeptraum und Notationen

Marc Born

Fraunhofer-Institut für Offene Kommunikationssysteme

Kaiserin-Augusta-Allee 31
10589 Berlin

born@fokus.fhg.de
Tel. +49 (30) 3463 7235
Fax. +49 (30) 3463 8235

VORWORT

Die Unterstützung der industriellen Fertigung von verteilten Telekommunikationssoftwaresystemen ist ein aktueller Forschungsschwerpunkt und zentrales Anliegen dieser Arbeit. Dabei konzentrieren sich die Untersuchungen auf die Integration von objektorientierten Modellierungstechniken und Komponentenarchitekturen mit dem Ziel einer automatischen Ableitung von Softwarekomponenten aus Entwurfsmodellen. Den Fokus von *CoRE* bildet die präzise Definition adäquat einsetzbarer Entwicklungstechniken und ihre Integration.

Im Band I dieser Arbeit [CoRE I] wurde bereits die Ausgangslage eingehend analysiert und diskutiert. Im Ergebnis dessen konnte festgestellt werden, daß sowohl die Weiterentwicklung und Spezialisierung von Softwareentwicklungsprozessen als auch die Integration von Entwicklungstechniken in diese Prozesse für die Unterstützung einer industriellen Softwarefertigung von außerordentlicher Relevanz sind. Für den Bereich telekommunikationsspezialisierter Entwicklungstechniken ist jedoch die momentane Situation trotz vielfältiger intensiver und aufwendiger Bemühungen industrieller und akademischer Einrichtungen nicht zufriedenstellend. Demzufolge konzentrieren sich die *CoRE*-Arbeiten gerade auf die Definition von Entwicklungstechniken und deren Integration für die industrielle Softwareentwicklung in diesem Bereich. Dabei sollen sich die vorgeschlagenen und erarbeiteten Entwicklungstechniken insbesondere in verschiedene konkrete und praxisrelevante Softwareentwicklungsprozesse einbinden lassen.

Von außerordentlicher Bedeutung ist in diesem Zusammenhang die präzise semantische Fundierung von *CoRE* in Form der Definition eines Konzeptraumes, dessen Kernkonzepte und wechselseitige Beziehungen auf der Basis einer detaillierten Anforderungsanalyse bereits informal eingeführt und diskutiert wurden. Die Präzisierung dieser Konzepte und deren Beziehungen sowie die notwendige formalisierte Konstruktion des Konzeptraumes $CoRE_{CEPT}$ sind Grundanliegen dieses Bandes. Dabei werden zur Bewertung der erarbeiteten Ergebnisse die in Abschnitt 1.5 von [CoRE I] formulierten Anforderungen, soweit sie für $CoRE_{CEPT}$ und $CoRE_{TATIONS}$ relevant sind, zugrundegelegt.

Auf der Basis von $CoRE_{CEPT}$ ist die Entwicklung der anderen in *CoRE* enthaltenen Entwicklungstechniken $CoRE_{TATIONS}$, $CoRE_{MAP}$ und $CoRE_{WARE}$ möglich. Die Formalisierung von $CoRE_{CEPT}$ bedingt die Auswahl einer geeigneten Technologie aus der Vielzahl möglicher Kandidaten. In diesem Band wird gezeigt, daß *Meta*

Object Facility (MOF, [OMG MOF 1.3]) eine geeignete Technologie ist. Die Konstruktion von $CoRE_{CEPT}$ erfolgt durch ihre Anwendung.

Die Präzisierung von $CoRE_{CEPT}$ verwendet eine Reihe von Konzepten weit verbreiteter Technologien und internationaler Standards als Ausgangspunkt. Die in diesem Zusammenhang bestehende besondere Bedeutung von RM-ODP [ITUT X.902] wird durch vergleichende Betrachtungen von $CoRE_{CEPT}$ und den entsprechenden Konzepten des RM-ODP herausgestellt. Weiterhin wird nachgewiesen, daß Telekommunikationssoftwaresysteme durch Entwurfsmodelle, die auf der Basis von $CoRE_{CEPT}$ definiert wurden, nicht nur schlechthin beschreibbar sind, sondern insbesondere die in [CoRE I] präzisierten Anforderungen an solche Systeme erfüllen, wobei die Forderung nach einer konzeptionellen Erweiterbarkeit der Entwicklungstechniken selbst berücksichtigt bleibt.

Neben der konzeptionellen Fundierung von *CoRE* wird in diesem Band gezeigt, daß verschiedene Notationen zur Darstellung von Entwurfsmodellen definiert werden können und - basierend auf $CoRE_{CEPT}$ - miteinander integrierbar sind. Die exemplarische Definition von zwei Notationen in $CoRE_{TATIONS}$ unterstreicht den sekundären Charakters von Notationen bezüglich der Entwicklungstechnik objektorientierte Modellierung mit dem in *CoRE* verfolgten Ansatz. Die Definition der Notationen erfolgt durch die Festlegung der Darstellung der konkreten Elemente von $CoRE_{CEPT}$. Kriterien zur Bewertung und Auswahl von Notationen im Kontext von *CoRE* werden in diesem Band ebenfalls formuliert.

Der Konzeptraum von *CoRE* bildet die Grundlage der angestrebten Integration der Entwicklungstechniken objektorientierte Modellierung, Einsatz von Komponentenarchitekturen und automatische Ableitung von Softwarekomponenten. Konkret sind die in [CoRE III] definierten Ableitungsregeln zur automatischen Ableitung von Softwarekomponenten gerade so definiert, daß alle Konzepte des Konzeptraumes und deren Beziehungen auf konkrete Mechanismen einer Komponentenarchitektur abgebildet werden. Die Möglichkeit der Definition dieser Ableitungsregeln und deren Realisierung in Form von Entwicklungswerkzeugen weisen zudem die Überführbarkeit von auf $CoRE_{CEPT}$ basierenden Entwurfsmodellen in konkrete Softwarekomponenten nach - ein Kriterium für die Güte eines Konzeptraumes.

In diesem Band wurden folgende Schriftarten eingesetzt:

- ***Elemente von Modellen sind in "Arial Italic" notiert,***
- **die Definition von Grammatikregeln der definierten textuellen Notation erfolgt in "Georgia Bold",**
- **Beispiele für die textuelle Notation werden in "Arial Bold" verfaßt und**
- *englische Begriffe, für die keine adäquate Übersetzung gefunden wurde, sind kursiv geschrieben.*

Berlin, im Juli 2001

Marc Born
Fraunhofer-Institut für Offene Kommunikationssysteme
Kaiserin-Augusta-Allee 31
10589 Berlin
born@fokus.fhg.de

INHALT

KAPITEL 1

Einführung in CoRE$_{CEPT}$ und CoRE$_{TATIONS}$

Der in dieser Arbeit konstruierte Konzeptraum *CoRE$_{CEPT}$* ist die Gesamtheit aller unter Benutzung von *CoRE* zur Modellierung von verteilten Softwaresystemen im Telekommunikationsbereich einsetzbaren Konzepte und ihrer Beziehungen. Damit wird das Instrumentarium definiert, das für die Beschreibung konkreter Entwurfsmodelle innerhalb von *CoRE* zur Verfügung steht. Dieses Instrumentarium ist die konzeptionelle Fundierung sowohl der Entwicklung konkreter Notationen innerhalb von *CoRE$_{TATIONS}$* als auch der Definition von Ableitungsregeln in *CoRE$_{MAP}$* für die automatische Ableitung von Softwarekomponenten aus Entwurfsmodellen.

Zur Festlegung dieses Instrumentariums wird hier eine geeignete Technik ausgewählt und angewendet. Dabei ist zu gewährleisten, daß:

- alle als notwendig identifizierten Konzepte und Beziehungen (vgl. [CoRE I], Kapitel 3) vollständig erfaßt sind,
- das Instrumentatrium minimal ist, d.h. daß alle definierten Konzepte und Relationen auch benötigt werden, um die aufgestellten Anforderungen an *CoRE* zu erfüllen,
- die Beschreibung des Instrumentariums offen ist für eventuelle Erweiterungen - sollen z.B. Konzepte zur Beschreibung des internen Verhaltens von CO-Typen in den Konzeptraum integriert werden, so soll diese Integration als Erweiterung der vorhandenen Definitionen möglich sein und nicht eine Neudefinition des Konzeptraumes erforderlich machen - diese Anforderung ergibt sich aus der in [CoRE I], Kapitel1 aufgestellten Forderung nach konzeptioneller Offenheit,
- die Beschreibung auf eine Art und Weise erfolgt, die eine einfache Möglichkeit für den Aufbau einer Werkzeugunterstützung für die Modellierung gestattet - diese Anforderung ergibt sich ebenfalls aus der in [CoRE I], Kapitel 1 aufgestellten Anforderung nach Werkzeugunterstützung.

Die Konstruktion von *CoRE$_{CEPT}$* erfolgt durch die Anwendung der objektorientierten Modellierung - das resultierende Modell aller Konzepte und Beziehungen wird als Metamodell von *CoRE* bezeichnet, jedes für die Modellbildung unter Benutzung von *CoRE* verwendbare Konstrukt ist im Metamodell erfaßt.

Ausgangspunkt für die Modellierung mit dem Ziel der Definition eines Metamodells sind alle Phänomene der Anwendungsdomäne, deren konkrete Ausprägungen in Entwurfsmodellen erfaßt werden sollen. In Metamodellen werden diese Phänomene klassifiziert und ihre für die Modellierung relevanten Eigenschaften und Beziehungen zu anderen Phänomenen spezifiziert.

> *(Beispiel 1)* Die Interaktionen zwischen Softwarekomponenten eines verteilten Softwaresystems erfolgen über Interfaces. Ein Entwurfsmodell enthält Beschreibungen aller Interfaces des modellierten Softwaresystems. Das Phänomen Interface ist somit in das Metamodell aufzunehmen, damit konkrete Interfacebeschreibungen in Entwurfsmodellen vorgenommen werden können. Dabei definiert das Metamodell u.a., daß konkrete Ausprägungen des Phänomens Interface in Entwurfsmodellen die Eigenschaft besitzen, identifizierbar zu sein, d.h. einen Namen zu tragen.

Durch die Verwendung objektorientierter Paradigmen zur Definition von Metamodellen ist deren spätere Erweiterbarkeit und Adaptierbarkeit weitestgehend gewährleistet. In einem objektorientierten Metamodell können Erweiterungen und Anpassungen durch Spezialisierungen von vorhandenen Metamodellkonstrukten erfolgen.

Auf der Basis von wohldefinierten Metamodellen können Entwurfsmodelle unabhängig von konkreten Notationen repräsentiert, ausgetauscht und verarbeitet werden. Der Grund für diese Flexibilität besteht darin, daß Metamodelle die automatische Ableitung von Repositorien gestatten, z.B. in Form von Datenbanken, die für die Speicherung von Entwurfsmodellinformationen geeignet sind. Weiterhin können aus Metamodellen automatisch Interfaces zum Zugriff und zur Manipulation von Entwurfsmodellinformationen, die in solchen Repositorien enthalten sind, erzeugt werden. Repositorien und ihre Interfaces werden insbesondere zur technologischen Realisierung der automatischen Ableitung von Softwarekomponenten benötigt. In Abschnitt 3.8 wird detaillierter auf die prototypische Realisierung im Rahmen dieser Arbeit eingegangen.

Eine Charakterisierung und eine Motivation der Konzepte von *CoRE$_{CEPT}$* in Bezug auf die Anwendungsdomäne Telekommunikation wurde bereits in [CoRE I], Kapitel3 vorgenommen. Ziel dieses Bandes ist die Präzisierung und Vervollständigung dieser Konzepte und ihrer Beziehungen durch die Definition eines geeigneten Metamodells. Insbesondere werden dabei alle relevanten Eigenschaften der Konzepte erfaßt und Regeln definiert, die bei der Definition von Entwurfsmodellen unter Benutzung dieser Konzepte beachtet werden müssen.

Solche Regeln werden auch als „*well formedness rules*“ bezeichnet, ihre Beachtung führt zu Entwurfsmodellen, die im Sinne der Anwendung der Modellierungskonzepte korrekt formuliert sind. Diese Eigenschaft garantiert aber noch nicht, daß die Ausführung der aus dem Entwurfsmodell abgeleiteten Softwarekomponenten auch den gewünschten Systemzweck erbringt. Untersuchungen diesbezüglich können nur erfolgen, wenn das Entwurfsmodell eine komplette Beschreibung des Verhaltens der die CO-Typen realisierenden Artefakte enthält - darauf wurde jedoch aus den in [CoRE I], Kapitel3 beschriebenen Gründen (vorerst) verzichtet.

Neben der Definition der konzeptionellen Grundlagen von *CoRE* besteht ein weiteres wesentliches Ziel dieses Bandes in der exemplarischen Definition von Notationen. Die Anwendung der Entwicklungstechnik Modellierung zur Lösung einer spezifischen Entwicklungsaufgabe für ein Softwaresystem innerhalb einer Anwendungsdomäne setzt die Existenz einer Notation für die Darstellung des resultierenden Entwurfsmodells voraus. Ein wesentlicher Vorteil von *CoRE* gegenüber bisher verwendeten Ansätzen liegt darin, flexibel bei der Auswahl dieser Notation sein zu können. Es ist möglich, statt einer einzigen Notation für alle Konzepte mehrere Notationen, die jeweils zur Darstellung bestimmter Konzepte und Beziehungen des Konzeptraumes besonders gut geeignet sind, kombiniert einzusetzen.

Eine Notation ist für die Darstellung bestimmter Aspekte gut geeignet, wenn ein notiertes Entwurfsmodell die entsprechenden realen Sachverhalte bezüglich der dazugehörigen Modellierungskonzepte vollständig, kompakt und leicht lesbar wiedergibt. Diese Charakterisierung ist offensichtlich hinsichtlich der Kompaktheit und leichten Lesbarkeit subjektiv, denn diese Kriterien hängen von dem Profil des jeweiligen Entwicklers ab. So ist es zwar allgemein anerkannt, daß eine graphische Darstellung die Lesbarkeit und das Verständnis für ein notiertes Entwurfsmodell erhöht, allerdings steigt bei hoher Komplexität eines Ent-

wurfsmodells auch die zur Präsentation erforderliche Darstellungsfläche stark an. Das hat zur Folge, das u.U. eine textuelle Darstellung übersichtlicher und damit besser zu erfassen und zu analysieren ist als eine graphische. Insbesondere bei graphischen Notationen hängt deren Eignung daher nicht nur von der Notation selbst ab, sondern auch von den Werkzeugen, die die Notation unterstützen. Werkzeuge sollten z.B. die Navigation zwischen zusammengehörigen Modellelementen ermöglichen, die nicht im gleichen Ausschnitt des graphisch notierten Entwurfsmodells sichtbar sind.

Die exemplarische Definition von zwei Notationen (graphisch und textuell) erfolgt in diesem Band nach einer Analyse der an eine Notation für *CORE* zu stellenden Anforderungen. Es wird gezeigt, daß sich im Falle der graphischen Notation eine Erweiterung einer existierenden graphischen Notation (*Unified Modeling Language*, UML [OMGUML1.3]) anbietet, da für diese bereits eine umfangreiche Werkzeugunterstützung verfügbar ist.

KAPITEL 2

Meta Object Facility

Eine generische Technologie zur Unterstützung der Definition und der automatischen Verarbeitung von Modellen und deren Metamodellen bietet der OMG-Standard *Meta Object Facility* (MOF) [OMG MOF1.3]. Der entscheidende Vorzug dieser Technologie ist die Bereitstellung objektorientierter Paradigmen zur Definition von Metamodellen und die Präsenz von Regeln, die die automatische Ableitung von Interfaces für Repositorien ermöglichen. Weiterhin sind auf der Grundlage des MOF-Standards eine Vielzahl von Werkzeugen für Modellrepräsentation, -transformation und programmiersprachliche Abbildung verfügbar. Damit lassen sich die eingangs aufgestellten Anforderungen an die Definition des Konzeptraumes für *CORE* hinsichtlich Erweiterbarkeit und Werkzeugunterstützung bereits durch die Auswahl dieser Technologie erfüllen. Die Forderungen nach Vollständigkeit und Minimalität hingegen sind durch die Definition des Metamodells selbst zu realisieren, ihre Erfüllung wird in Kapitel3 durch die MOF-basierte Konstruktion von $CORE_{CEPT}$ gezeigt.

Um ein Metamodell durch objektorientierte Modellierung definieren zu können, muß ein weiteres Modell existieren, das das begriffliche Instrumentarium zur Definition von Metamodellen festlegt. Dieses Modell (Meta-Metamodell) enthält diejenigen Konzepte und Beziehungen, die zur Definition eines Metamodells einsetzbar sind (vgl. Abb. 1).

> *(Beispiel 2)* Sei ***"Dies ist ein Beispiel"*** ein konkretes Informationselement im Kontext einer Anwendung (*User-Object*-Ebene). Dann instanziiert dieses Informationselement das Modellelement ***BeispielString*** (*Model*-Ebene). Das Modellelement ***BeispielString*** instanziiert wiederum das Metamodellelement ***string*** (*Meta-Model*-Ebene). Letztendlich instanziiert das Metamodellelement ***string*** das Meta-Metamodellelement ***primitive*** als Basis für primitive Datentypen (*Meta-Meta-Model*-Ebene).

Der Abstraktionsgrad steigt ausgehend von der *User-Object*-Ebene in Richtung Meta-Metamodell an, wobei i.allg. die Anzahl der Modellelemente der jeweiligen Ebene abnimmt. Aus diesem Grund wird die Abstraktionskette nicht über die Meta-Metamodellebene hinaus weitergeführt, sondern mit der Definition elementarer Konzepte auf diesem Niveau beendet.

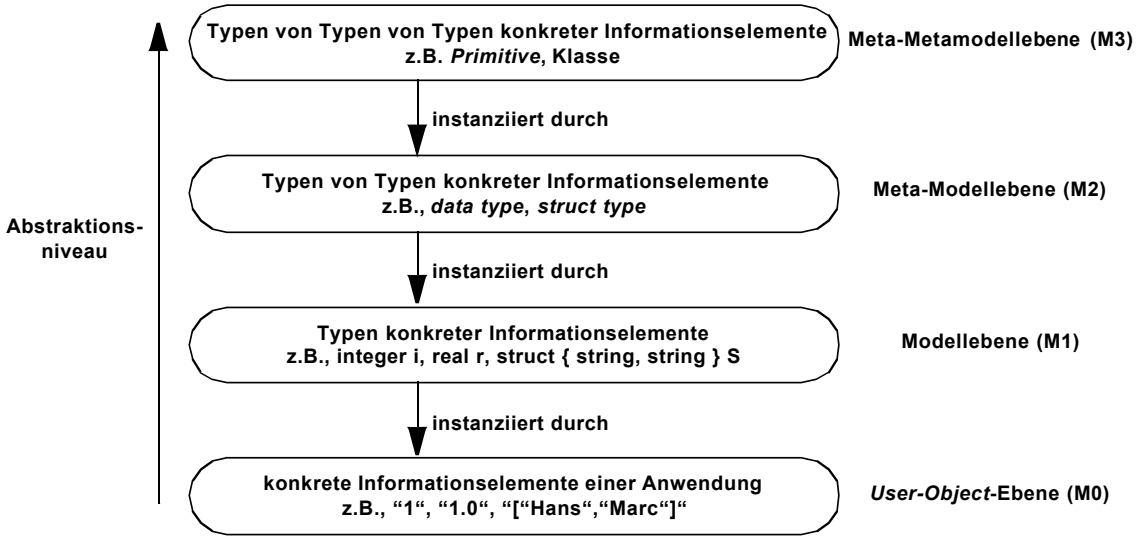


Abb. 1 Meta-Metamodell, Metamodell und Modell

Der Standard *Meta Object Facility* definiert:

- eine Menge von Meta-Metamodellkonzepten (ein Meta-Metamodell), die zur Definition konkreter, MOF-konformer Metamodelle[1] einsetzbar ist,
- Ableitungsregeln für die Erzeugung von CORBA-IDL-Interfaces zum Zugriff auf Repositorien für Modelle aus MOF-konformen Metamodellen sowie
- Regeln zu Erzeugung von XML-Dokumentenformaten (XML-DTD) aus MOF-konformen Metamodellen zum Modellaustausch (enthalten in [OMGXMI1.1]).

Die folgenden Meta-Metamodellkonstrukte aus MOF kommen in dieser Arbeit für die Definition des Metamodells zum Einsatz:

- ***data type***
  Die Definition der Metamodellelemente erfordert häufig die Präsenz von Grunddatentypen wie ***String*** oder ***Integer*** im Metamodell. Diese dienen zur Definition von anderen Konzepten oder deren Eigenschaften, die z.B. auf ***attribute*** und ***operation*** basieren. Dazu wird das Konstrukt ***data type*** als Meta-Metamodellkonstrukt durch MOF definiert. ***Data type*** wird benutzt für die Repräsentation von Datentypen, deren Werte keine Objektidentität haben.
  Zur Vereinfachung wird für Datentypen im Metamodell von *CoRE* ausschließlich das CORBA-IDL-Typsystem benutzt, da es alle hier relevanten Grunddatentypen bereitstellt.
- ***class***
  Klassen dienen zur Beschreibung von Typen von Modellelementen der M1-Ebene auf der M2-Ebene. Instanzen dieser so beschriebenen Typen treten demzufolge auf der M1-Ebene auf, also in konkreten Modellen. Diese Instanzen haben Identität, Zustand und Verhalten, wobei ihre Zustandsinformationselemente durch die Klassenbeschreibung im Metamodell definiert sind und das Verhalten durch die MOF-Spezifikation impliziert wird. Dieses Verhalten wird durch Repositorien realisiert, die auf der Grundlage des MOF-Standards automatisch aus der Metamodelldefinition abgeleitet werden können.
  Klassendefinitionen beinhalten Attribute und Operationen zur Definition ihres Zustandes und der Zugriffsmöglichkeiten auf ihre Instanzen. Klassen dürfen in einer Vererbungsrelation stehen. Die Defini-

1. Der Terminus „konform zu MOF" bezieht sich auf die Einhaltung der Konformitätsaussagen in Abschnitt 0.6 von [OMGMOF1.3], speziell *Compliance Point One*.

tion einer solchen Vererbungsrelation bedeutet (wie in anderen objektorientierten Techniken), daß der gesamte Inhalt der Superklasse auch in der abgeleiteten Klasse zur Verfügung steht sowie daß alle Regeln (*constraints*) , die für die Superklasse definiert wurden, auch für die abgeleitete Klasse gelten. Der MOF-Standard erlaubt ausdrücklich die Mehrfachvererbung, d.h. eine Klasse darf mehrere Superklassen besitzen.

- ***attribute*** und ***operation***
  Ein Attribut dient zur Aufnahme von Werten eines bestimmten Typs in Instanzen der Klasse, die das Attribut enthält. Ein Attribut besitzt einen innerhalb der definierenden Klasse eindeutigen Namen, einen Typ, der eine andere Klasse oder ein Datentyp sein kann, sowie eine Spezifikation über Zugriffs- und Modifikationsrechte (z.B. *readonly*) eines Attributes.
  Operationen spezifizieren, wie auf das Verhalten der definierenden Klasse zugegriffen werden kann. Über eine Operationsdefiniton wird ausschließlich die Signatur festgelegt, unter Benutzung auf das Verhalten der die Operation definierenden Klasse zugegriffen werden kann, aber nicht, wie dieses Verhalten erbracht wird. Operationen besitzen einen im Kontext der definierenden Klasse eindeutigen Namen, eine Liste von Parametern sowie einen optionalen Typ für einen Rückagabewert. Dieser Typ kann eine Klasse oder ein Datentyp sein.
  Parameter besitzen wiederum einen Namen, einen Typ (Klasse oder Datentyp) sowie eine Spezifikation, in welche Richtung der Parameter bei einem Operationsaufruf zu übergeben ist (vom Rufer zum Gerufenen, vom Gerufenen zum Rufer oder in beide Richtungen).
  Die Signatur einer Operation darf eine Liste von Ausnahmen beinhalten, die bei der Ausführung der Operation ausgelöst werden können.
- ***abstract class***
  Klassen können als abstrakte Klassen definiert werden. Solche Klassen dienen der Verwendung in Vererbungshierarchien, es existieren aber niemals Instanzen (also Modellelemente) die als spezialisiertesten Typ eine abstrakte Klasse besitzen.
- ***association***
  Assoziationen werden zur Modellierung von Beziehungen zwischen Klassendefinitionen in einem Metamodell verwendet. In einem Modell können damit konkrete Beziehungen zwischen Instanzen solcher Klassen erfaßt werden. Diese Beziehungen besitzen allerdings keine Identität, folglich können keine Attribute oder Operationen für Assoziationen definiert werden.
  Jede Assoziation besitzt genau zwei Endpunkte, wobei jeder dieser Endpunkte die folgenden Eigenschaften besitzt:
    - einen (innerhalb der Assoziation eindeutigen) Namen,
    - einen Typ, d.h. eine Klassendefinition, die in die Assoziation an diesem Endpunkt involviert ist,
    - eine Definition der Vielfachheit der Assoziation (d.h. wieviele Instanzen des Typs des Endpunktes dürfen zu einer Instanz des Typs des anderen Endpunktes in Relation stehen,
    - eine Spezifikation, ob zu diesem Endpunkt der Assoziation navigiert werden kann, d.h. ob Referenzen für diesen Endpunkt aus der Perspektive des anderen Endpunktes definiert werden,
    - eine Spezifikation, ob es sich aus der Perspektive dieses Endpunktes um eine Aggregation handelt.

  In MOF werden zwei Arten von Aggregationen unterstützt, die Komposition und die lockere Aggregation (*loose aggregation*). Die lockere Aggregation besitzt keine Einschränkungen bezüglich der Vielfachheitsspezifikationen ihrer Endpunkte, d.h. der Menge der Instanzen an den Endpunkten in konkreten Ausprägungen der Relation und des Lebenszyklus dieser Instanzen. Im Gegensatz dazu sind bei der Komposition die komponierten Teile als Bestandteile des Ganzen zu verstehen, was bedeutet, daß ihre

Instanzen immer *genau einer* Instanz der Komposition zuzuordnen sind und daß ihre Lebenszeit auf die Lebenszeit der Komposition beschränkt ist.

- ***package***
  Pakete dienen der Strukturierung von Metamodellen. Auf der M2-Ebene können Pakete andere Pakete, Klassen, Assoziationen, Datentypen und Ausnahmen beinhalten.
- ***constraint***
  Die Verwendung der in einem Metamodell definierten Konstrukte zur Definition von Modellen muß unter Beachtung bestimmter Regeln erfolgen. Diese Regeln sind zusätzlich zum eigentlichen Metamodell definiert, sie beziehen sich aber stets auf die im Metamodell spezifizierten Konstrukte. Die Einhaltung der Regeln führt zu korrekt spezifizierten Modellen.

Sind Metamodelle ausschließlich auf der Basis der in MOF enthaltenen Meta-Metamodellkonstrukte beschrieben, so lassen sich IDL-Interfaces für den Zugriff auf bzw. die Manipulation von Modellen, die auf diesen Metamodellen basieren, automatisch erzeugen. Dabei wird ausgenutzt, daß das Metamodell alle Informationen über die Modellkonstrukte enthält und durch die Verwendung einer kleinen Menge von Meta-Metamodellelementen auch feststeht, in welcher Struktur diese Information vorliegt. Die Erzeugung einer Implementierung für die automatisch erzeugten Interfaces ist ebenfalls automatisch möglich, allerdings ist ein Verfahren für die Generierung solcher Implementierungen nicht standardisiert sondern den Herstellern von entsprechenden Produkten (z.B. [DSTC dMOF]) überlassen.

Sollen Modelle zwischen verschiedenen Repositorien oder anderen Implementierungen wie z.B. Modellierungswerkzeugen ausgetauscht werden, so bieten sich zwei Möglichkeiten an:

- Der Austausch erfolgt durch die Benutzung der aus dem Metamodell erzeugten CORBA-IDL-Interfaces. Dabei muß die gesamte Information aber u.U. mühsam durch wiederholte Operationsrufe an verschiedenen Interfaces bereitgestellt werden.
- Der Austausch erfolgt auf der Basis von XML-Dokumenten [W3CXML]. Das Format für diese XML-Dokumente ist durch eine XML-DTD (*Document-Type*-Definition) vorgegeben, diese wird automatisch aus dem Metamodell erzeugt.

Das zweite Verfahren ist eine sehr elegante Möglichkeit, um den Modellaustausch von vollständigen Modellen oder Teilmodellen technologieunabhängig zu ermöglichen. Technologieunabhängigkeit bedeutet an dieser Stelle, daß XML-Dateien Textdokumente sind, zu deren Austausch verschiedene Technologien eingesetzt werden können. Dabei kann zusätzlich zum eigentlichen Modell auch noch dessen Metamodell ausgetauscht werden. Dieses wird benötigt, um die in einem Modell enthaltene Information auch interpretieren zu können, wenn auf der empfangenden Seite das Metamodell nicht bekannt ist. Ermöglicht wird der Austausch von Metamodellen durch die Tatsache, daß die Konzepte zur Definition von Metamodellen durch das Meta-Metamodell festgelegt sind und die Generierungsregeln für XML-DTDs auch für das Meta-Metamodell selbst angewendet werden können. XML-Dokumente, die der DTD genügen, die aus dem Meta-Metamodell von MOF erzeugt wurde, stellen gerade MOF-konforme Metamodelle dar. Diese Situation ist in Tab. 1 dargestellt. Auf der Meta-Metamodellebene (M3) ist das MOF Meta-Metamodell definiert. Dieses Meta-Metamodell ist auch Basis einer XML-DTD für MOF.

Metamodelle (M2) lassen sich unter Verwendung der Beschreibungsmittel des MOF-Meta-Metamodells definieren und als XML-Dokumente repräsentieren, die der MOF XML-DTD genügen. Der MOF-Standard definiert, wie sich für Metamodelle XML-DTDs erzeugen lassen.

Modelle (M1) lassen sich mit den in einem Metamodell fixierten Konzepten formulieren und in XML-Dokumenten repräsentieren. Diese XML-Dokumente genügen der XML-DTD, die aus dem Metamodell erzeugt wurde.

MOF wird im Kontext von *CORE* zur Definition des Konzeptraumes in Form eines Metamodells eingesetzt.

Meta-ebene	Modell	XML-DTD	XML-Dokument
M3	MOF Meta-Metamodell	MOF XML-DTD	
M2	MOF konformes Metamodell	XML-DTD für Metamodell	XML-Dokument für Metamodell entsprechend. MOF XML-DTD
M1	Modell entsprechend Metamodell		XML-Dokument für Modell entsprechend XML-DTD für Metamodell
M0	Instanzen von Modellelementen		

Tab.1 Modelle, XML-DTDs und XML-Dokumente in MOF

KAPITEL 3

Definition des Metamodells von CoRE$_{CEPT}$

Die im vorangegangenen Kapitel vorgestellte Technologie *Meta Object Facility* wird nun zur Definition von *CoRE$_{CEPT}$* benutzt. Zur Notation des Metamodells von *CoRE$_{CEPT}$* werden Klassendiagramme des UML-Standards eingesetzt. Die graphische Notation erleichtert das Verständnis des Metamodells, da die oben aufgeführten MOF-Meta-Metamodellelemente direkt zu Konstrukten der UML-Klassendiagramme korrespondieren (vgl. Abb.2). Die Verwendung von UML wird auch durch den MOF-Standard nahegelegt, eine Normierung der Nutzung von UML zur Definition von MOF-konformen Metamodellen ist gegenwärtig in Vorbereitung [OMG MOFP].

Zusätzlich zu den MOF-Meta-Metamodellelementen wird in dieser Arbeit die unidirektionale Relation ***Dependency*** verwendet, um Abhängigkeiten zwischen Metamodellelementen zu symbolisieren. Instanzen dieser Relation dienen nur zur Illustration, sie haben keine Bedeutung für die Ableitung von CORBA-IDL-Interfaces, Repositorien oder XML-DTDs aus dem Metamodell. Die Semantik einer ***Dependency***-Definition in einem Metamodell ist analog zu der Semantik einer ***Dependency***-Definition in einem UML-Modell, d.h. sie legt fest, daß Veränderungen des *Supplier*-Endpunktes der ***Dependency***-Definition Auswirkungen auf den *Client*-Endpunkt haben.

Eine alternative Möglichkeit zur Verwendung von UML als Notation für das Metamodell ist der Einsatz der textuellen Sprache *Meta Object Definition Language* (MODL [DSTCdMOF]). An dieser Stelle wird auf eine MODL Spezifikation jedoch verzichtet, da nur eine kommerziell verfügbare Werkzeugumgebung zur automatischen Ableitung von Repositorien und Interfaces, die auf der MODL-Sprache basiert, existiert. Darüber hinaus ist eine Standardisierung von MODL im Gegensatz zum UML-Profil für MOF nicht vorgesehen.

Andere Werkzeugumgebungen sind ausschließlich auf die Verwendung von UML-Beschreibungen von Metamodellen ausgelegt. Außerdem ist die Ausdrucksfähigkeit der UML-Klassendiagramme und der MODL-Konstrukte in Bezug auf die hier verwendeten Meta-Metamodellelemente identisch, so daß sich Spezifikationen in MODL in UML-Klassendiagramme überführen lassen - und umgekehrt.

Ein zentraler Bestandteil einer MOF-konformen Spezifikation eines Metamodells ist die Beschreibung der Regeln (*Constraints*), die für die Anwendung der im Metamodell enthaltenen Elemente zur Modellbildung gel-

ten sollen. Diese Regeln legen fest, wie Modelle formuliert werden müssen, um im Sinne des Metamodells korrekt zu sein. Die Regeln sind durch Implementierungen eines Metamodells (*Repository*) sicherzustellen. Die Regeln des Metamodells von *CoRE* werden als Text formuliert und beziehen sich auf die Metamodellelemente von *CoRE*.

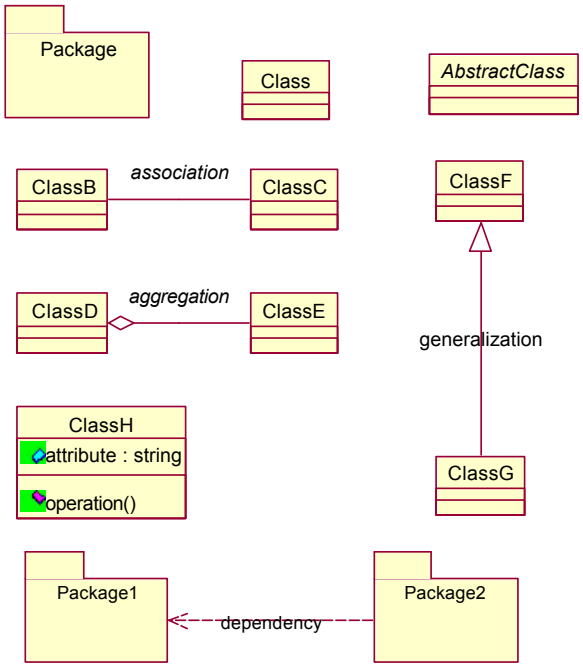


Abb. 2 Notation von Metamodellen mit UML

3.1 Sichten im Metamodell

Die Definition des Metamodells von *CoRE* erfolgt entsprechend der in [CoRE I], Kapitel 3 vorgenommenen Strukturierung der Konzepte in Sichten.

Die Sichten werden mit dem Meta-Metamodellelement ***Package*** im Metamodell erfaßt. Für alle Sichten von *CoRE$_{CEPT}$* wird eine korrespondierende ***Package***-Definition im Metamodell eingeführt wobei der Name der ***Package***-Definition der Name der Sicht ist. Alle Konzepte, die der jeweiligen Sicht zugeordnet werden, sind in der korrespondierenden ***Package***-Definition des Metamodells definiert.

Zwischen den Konzepten, die den verschiedenen Sichten zugeordnet sind, bestehen diverse Abhängigkeiten. So ist z.B. *Port*-Definition als Bestandteil der Konfigurationssicht abhängig vom Konzept Interfacetyp aus der Struktursicht (Eine *Port*-Definition in einem Entwurfsmodell erfordert zwingend die Angabe eines Interfacetyps im Entwurfsmodell). Die Abhängigkeiten der Konzepte untereinander implizieren Abhängigkeiten der Sichten, wobei gilt: Eine Abhängigkeit einer Sicht ***A*** von einer Sicht ***B*** besteht genau dann, wenn es ein in ***A*** enthaltenes Konzept ***KA*** gibt, das von einem Konzept ***KB*** abhängt, das in ***B*** enthalten ist.

In Abb.3 sind die Sichten und ihre Abhängigkeiten im Metamodell definiert. Aufgrund welcher Abhängigkeiten zwischen Konzepten die Abhängigkeiten zwischen den ***Package***-Definitionen zustande kommen, wird nachfolgend während der Konstruktion des Metamodells erläutert.

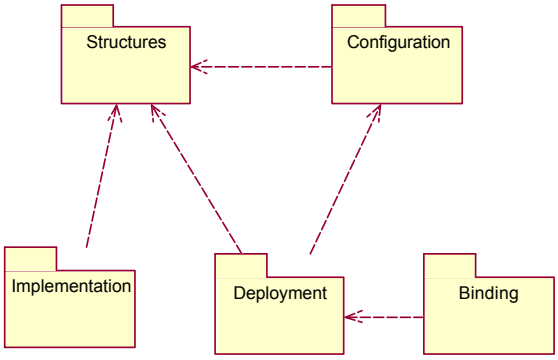


Abb. 3 Modellierung der Sichten im Metamodell

3.2 Struktursicht

Die Struktursicht enthält Konzepte, die zur Modellierung von CO-Typen und Interfacetypen in Verbindung mit den an den Interfacetypen vorhandenen Interaktionselementen geeignet sind. Wie bereits dargestellt, sind die in der Struktursicht enthaltenen Konzepte Datentyp, Namensraum, Operation, Ausnahme, Parameter und Interfacetyp mit ausschließlich operationalen Interaktionselementen in Anlehnung an die aus CORBA-IDL bekannten Konzepte definiert worden. Für diese Konzepte existiert bereits ein MOF-konformes Metamodell [OMG CCM II], das als Basis für die in der Struktursicht enthaltenen Erweiterungen (Signalinteraktion, *Continous-Media*-Interaktion, CO-Typen etc.) herangezogen werden kann. Da das Metamodell von *CORE* durch die Anwendung des MOF-Standards entsteht, läßt sich das bereits als MOF-konformes Modell beschriebene CORBA-IDL-Datentypmodell kanonisch integrieren. Das CORBA-IDL-Metamodell kann durch die Anwendung von Generalisierungsrelationen zu einem Metamodell erweitert werden, das den vollständigen Konzeptraum von *CORE* umfaßt.

Die Entscheidung, das CORBA-IDL-Metamodell als Basis für das zu definierende Metamodell einzusetzen, ist aus den folgenden Gründen getroffen worden:

- für das CORBA-IDL-Metamodell existieren eine Reihe von Implementierungen wie Repositorien oder *Compiler* - diese Implementierungen können für die notwendige Realisierung der Werkzeugunterstützung von *CORE* wiederverwendet und erweitert werden,
- bestehende CORBA-IDL-Spezifikationen basieren dann auf einem Teilmodell des Metamodells von *CORE*, können also wiederverwendet und mit den zusätzlichen Konzepten von $CORE_{CEPT}$ erweitert werden - Werkzeuge für *CORE* sind *a priori* für die Behandlung von CORBA-IDL-Spezifikationen geeignet,
- das Metamodell von CORBA-IDL enthält einige fundamentale Modellierungsmuster, die innerhalb der hier vorzunehmenden Erweiterung wiederverwendet werden können, dazu gehört u.a. das Muster von Containern, die andere Definitionen beinhalten (*Container-Contained*-Muster).

Das Metamodell von CORBA-IDL wird in das Metamodell von *CORE* aufgenommen, indem eine ***Package***-Definition ***IDL*** eingeführt wird, in der alle Metamodellelemente von CORBA-IDL enthalten sind. Zur besseren Strukturierung wird außerdem eine ***Package***-Definition ***AdvancedConcepts*** vorgenommen, die die für die Sichten von $CORE_{CEPT}$ definierten korrespondierenden ***Package***-Definitionen enthält. Aufgrund der Verwendung der Metamodellelemente von CORBA-IDL (z.B. Datentyp) für die im folgenden vorgenommenen Definitionen von Metamodellelementen wird eine Abhängigkeit zwischen den ***Package***-Definitionen ***AdvancedConcepts*** und ***IDL*** definiert. Diese Situation ist in Abb.4 illustriert.

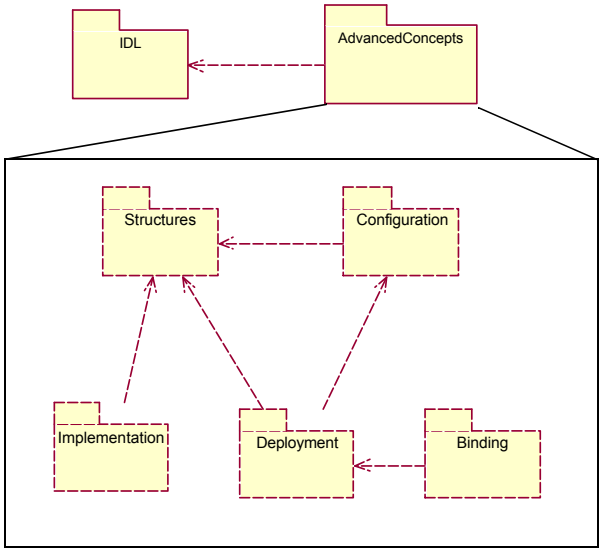


Abb. 4 Paketstruktur des Metamodells

3.2.1 Namensraum

Das Konzept Namensraum dient der Strukturierung von Entwurfsmodellen, da die notwendige Eindeutigkeit der Namen von Modellelementen durch ihre Zuordnung zu Instanzen des Konzeptes Namensraum erreicht wird. Dieses Verfahren soll für alle innerhalb von *CoRE$_{CEPT}$* definierten Konzepte anwendbar sein. Da das Konzept Namensraum bereits in CORBA-IDL definiert ist, existiert auch eine entsprechende Definition im Metamodell für CORBA-IDL, die hier wiederverwendet und erweitert wird. Diese Definition basiert auf einem wichtigen Modellierungsmuster des CORBA-IDL-Metamodells, der ***Container-Contained***-Assoziation. Diese Assoziation (genauer Komposition) ist in Abb. 5 dargestellt. Die Semantik besteht darin, daß in einem Entwurfsmodell in einer Instanz einer von ***Container*** abgeleiteten Klasse eine Menge von Instanzen einer von ***Contained*** abgeleiteten Klasse enthalten sind.

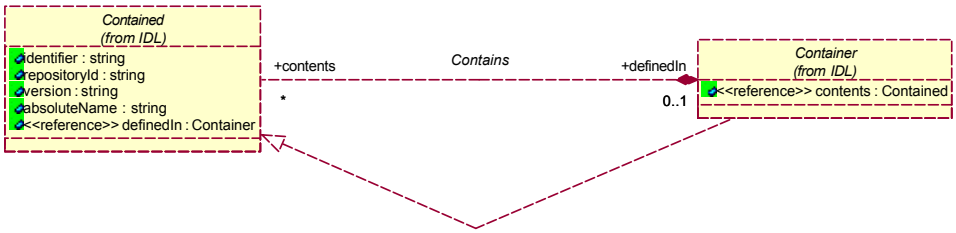


Abb. 5 *Container-Contained*-Assoziation aus dem CORBA-IDL-Metamodell

Container und ***Contained*** sind abstrakte Klassen im Metamodell wobei eine Generalisierungsrelation von ***Container*** zu ***Contained*** spezifiziert ist. Die Klasse ***Container*** besitzt eine Assoziation der Art Komposition zu der Klasse ***Contained***, d.h. Instanzen einer von ***Container*** abgeleiteten Klasse beinhalten eine Menge von Instanzen von Klassen, die von ***Contained*** abgeleitet sind.

Jede Instanz einer abgeleiteten Klasse der Klasse ***Contained*** in einem Entwurfsmodell besitzt einen Namen (modelliert durch das Attribut ***identifier***) sowie einen absoluten Namen (modelliert durch das Attribut ***absoluteName***). Der absolute Name wird gebildet durch Verkettung der Namen von ineinander verschachtelten Instanzen von abgeleiteten Klassen von ***Container*** und ***Contained*** (Jede ***Container***-Instanz ist auch eine

Contained-Instanz). Weiterhin sind die Attribute ***repositoryId*** und ***version*** definiert, diese erlangen bei der technologischen Umsetzung des Metamodells durch ein Repositorium für Entwurfsmodelle Bedeutung. Jeder Eintrag im Repositorium ist dann nicht nur durch seinen Namen sondern auch durch einen Bezeichner innerhalb des Repositoriums eindeutig identifizierbar. Das Attribut ***version*** dient zur Unterscheidung verschiedener Versionen ein und desselben Modellelements in einem Repositorium.

Die Klasse ***Container*** bietet die Basis für die Definitionen des Konzeptes Namensraum in dem Sinne, daß jede von ***Container*** abgeleitete Klasse einen Namensraum bildet. Bisher ist allerdings keine Klasse eingeführt, die tatsächlich in einem konkreten Entwurfsmodell instanziiert werden kann, da ***Container*** selbst als abstrakte Klasse modelliert wurde. Zu diesem Zweck wird in dem CORBA-IDL-Metamodell die Klasse ***ModuleDef*** als Ableitung von ***Container*** definiert. Diese hat keine über ***Container*** hinausgehende Eigenschaften, kann aber in konkreten Entwurfsmodellen instanziiert werden.

Um die Semantik des CORBA-IDL-Metamodells, konkret des Elements ***ModuleDef,*** nicht zu verändern, wird hier eine weitere Ableitung von ***ModuleDef*** in Form der Klasse ***NamespaceDef*** definiert. Diese dient in konkreten Entwurfsmodellen zur Aufnahme von Instanzen der im Metamodell von *CORE* neu eingeführten Klassen. Die Definition der Klasse ***NamespaceDef*** ist in Abb. 6 illustriert.

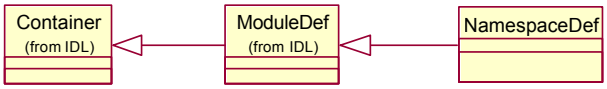


Abb. 6 Definition der Klasse *NamespaceDef*

Es muß gefordert werden, daß ***NamespaceDef***-Definitionen in einem Entwurfsmodell nur innerhalb von ***NamespaceDef***-Definitionen vorgenommen werden können, nicht aber in Instanzen anderer, von ***Container*** abgeleiteter Metaklassen. Diese Forderung dient zur Aufrechterhaltung der Semantik des Metamodells von CORBA-IDL.

(Constraint 1) Für Instanzen der Metaklasse ***NamespaceDef*** in einem Entwurfsmodell, für die eine konkrete ***Contains***-Assoziation definiert ist, muß die Instanz der assoziierten, von ***Container*** abgeleiteten Klasse ebenfalls vom Meta-Typ ***NamespaceDef*** sein.

3.2.2 Datentyp, Interfacetyp, Operation, Attribut, Ausnahme

Das Metamodell von CORBA-IDL enthält alle Klassendefinitionen, deren Instanzen in konkreten Entwurfsmodellen gerade der Beschreibung von Datentypen, Operationen, Attributen (als spezielle Variante von Operationen im Sinne einer verkürzten Schreibweise für *get*- und *set*-Operationen für einen bestimmten Datentyp) und Ausnahmen dienen. Weiterhin ist die Definition von Interfaces mit rein operationaler Signatur erlaubt. Die genaue Spezifikation des CORBA-IDL-Metamodells ist in [OMG CCM II] zu finden und wird hier nicht näher erläutert.

In Abb. 7 ist das Metamodell aller aus CORBA-IDL bekannten Datentypen dargestellt. Hervorgehoben sei, daß alle Datentypdefinitionen von der abstrakten Klasse ***IDLType*** abgeleitet sind. Damit verbunden ist ein weiteres Modellierungsmuster, das im Rahmen der Definition des hier vorgestellten Metamodells wiederverwendet wird: Das *Type-Typed*-Muster. Dieses Muster wird realisiert durch die Einführung der abstrakten Klasse ***IDLType***, die als Basisklasse für alle Klassen dient, deren Instanzen in einem Entwurfsmodell als Typdefinitionen betrachtet werden. Dazu gehören (wie in Abb.7 dargestellt) alle CORBA-IDL-Datentypen, aber auch Interfacetypen. Instanzen der von ***IDLType*** abgeleiteten Klassen in einem Entwurfsmodell können als Rückgabetypen von Operationen, als Typen von Parameterdefinitionen sowie zur Definition von strukturierten Datentypen verwendet werden. Konzepte, die sich dabei auf genau einen Typ beziehen, werden

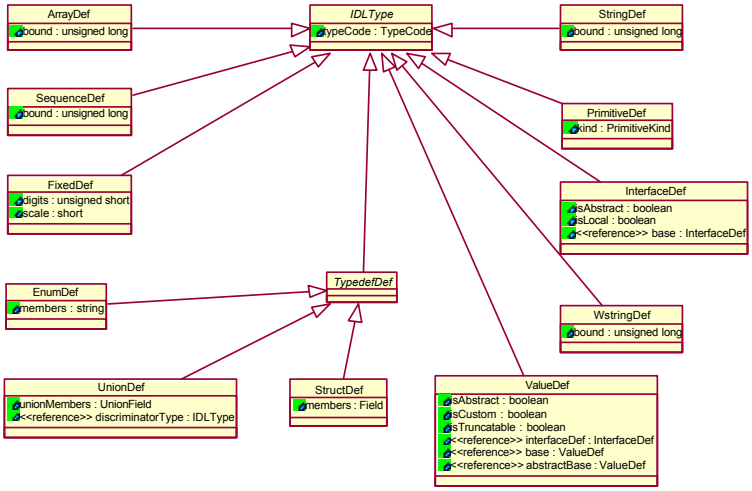


Abb. 7 Metamodell der Datentypen von CORBA-IDL

von der abstrakten Klasse ***Typed*** abgeleitet, für die eine Assoziation zu ***IDLType*** spezifiziert ist. So ist z.B. die Klasse ***OperationDef*** von ***Typed*** abgeleitet, da Operationen ja stets eine Relation zu einem Rückgabetyp haben (vgl. Abb. 8)[1].

Die Einführung der abstrakten Klasse ***IDLType***[2] hat noch einen weiteren Vorteil: Sollen andere Datentypmodelle in *CORE$_{CEPT}$* integriert werden, so sind deren Metaklassen als Ableitung von ***IDLType*** zu definieren. Dann dürfen in Entwurfsmodellen Instanzen der Metaklassen des alternativen Datentypmodells so verwendet werden, wie alle anderen Instanzen der von ***IDLType*** abgeleiteten Klassen. Voraussetzung ist die Beschreibung der Konzepte des alternativen Datentypmodells als MOF-konformes Metamodell, d.h. unter Verwendung der MOF-Meta-Metamodellelemente.

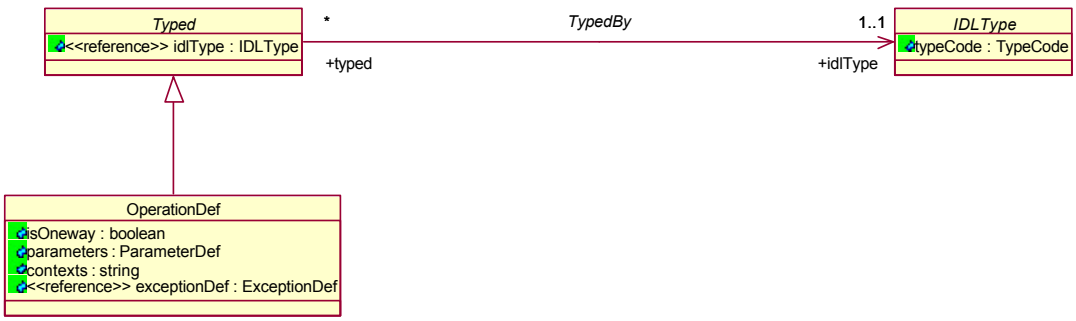


Abb. 8 Type-Typed-Muster des CORBA-IDL-Metamodells

1. Falls eine Operationsdefinition keinen Rückgabetyp spezifiziert, so wird der primitive Datentyp ***void*** verwendet.
2. Der Klassenname ***IDLType*** wurde aus dem CORBA-IDL-Metamodell übernommen, diese Klasse impliziert keine Abhängigkeiten von CORBA-IDL-Datentypen.

3.2.3 Signaltyp und Signalparameter

Ein wesentliches Ziel für die Definition von *CORE*$_{CEPT}$ liegt in der homogenen Unterstützung aller im Kontext von Telekommunikationssoftwaresystemen relevanten Interaktionsarten. Homogene Unterstützung bedeutet die konzeptionelle Gleichbehandlung der Interaktionselemente aller Interaktionsarten in Entwurfsmodellen. Die im CORBA-IDL-Metamodell enthaltene operationale Interaktion ist im Metamodell durch die Metaklasse ***OperationDef*** erfaßt, deren Bestandteile Parameterdefinitionen und durch diese referenzierte Datentypdefinitionen sind. Analog sind Signaltypen die elementaren Bestandteile der Signalinteraktionsart. Eine Signaltypdefinition in einem Entwurfsmodell legt dabei fest, wie konkrete Signale, die zur Ausführungszeit im Zuge einer Interaktion ausgetauscht werden, strukturiert sind und welche Eigenschaften sie aufweisen. Die Struktur von Signalen wird in einer Signaltypdefinition durch die identifizierbare Angabe der von Signalen zu transportierenden Signalparameter spezifiziert, wobei deren Werte gerade Instanzen von *Value*-Typdefinitionen im Entwurfsmodell sind. Ein Signalparameter in einem Entwurfsmodell referenziert also eine *Value*-Typdefinition im Entwurfsmodell.

Für das Konzept Signaltyp wird eine Metaklasse ***SignalDef*** in das Metamodell eingeführt (vgl. Abb. 9). Diese Metaklasse ist eine Ableitung der abstrakten Klasse ***IDLType***, damit ist die Einbeziehung von Signaltypdefinitionen in das grundlegende Modellierungsmuster *Type-Typed* möglich. Die Nützlichkeit dieses Modellierungsmusters in Bezug auf Signaltypdefinitionen wird bei der nachfolgenden Einführung der auf Signaltypen basierenden Interaktionselemente deutlich.

Die Metaklasse ***SignalDef*** ist gleichzeitig Ableitung der abstrakten Metaklasse ***Contained***, d.h. Instanzen von ***SignalDef*** müssen innerhalb eines geeigneten Containers im Entwurfsmodell definiert werden. Hier wird fest-

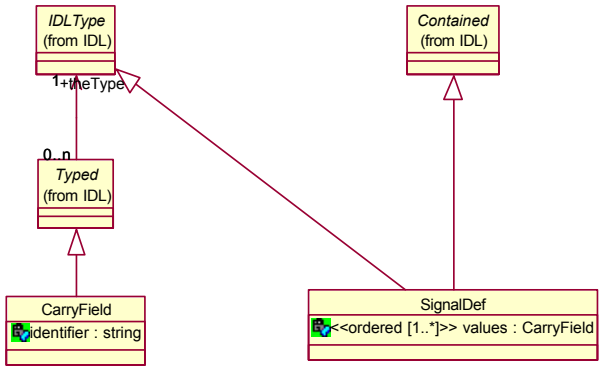


Abb. 9 Metamodell für Signaltyp und Signalparameter

gelegt, daß Container, in denen Signaltypen in einem Entwurfsmodell definiert werden dürfen, Instanzen der Metaklasse ***NamespaceDef*** sind. Damit wird sichergestellt, daß die Semantik des zugrundeliegenden CORBA-IDL-Metamodells nicht verändert wird, d.h. daß in Instanzen der Ableitungen von ***Container***, die im CORBA-IDL-Metamodell definiert sind, in Entwurfsmodellen keine Signaltypdefinitionen vorgenommen werden dürfen - hiermit wird die Semantik von ***SignalDef*** als ***Contained*** festgelegt.

(Constraint 2) Instanzen von ***SignalDef*** dürfen nur in Instanzen von ***NamespaceDef*** enthalten sein.

Die Relation von Signaltypen zu Signalparametern wird durch ein Attribut des Namens ***values*** der Metaklasse ***SignalDef*** im Metamodell erfaßt. Der Typ des Attributs ist eine geordnete, nichtleere Liste, wobei die Listenelemente den Typ ***CarryField*** besitzen. Die Metaklasse ***CarryField*** besitzt ein Attribut ***identifier*** vom Typ ***string***, wodurch die geforderte Möglichkeit der Identifikation der Signalparameter hergestellt wird. Die Metaklasse ***CarryField*** ist von der abstrakten Metaklasse ***Typed*** abgeleitet. Damit wird im Metamodell erfaßt,

daß jede konkrete Instanz von ***CarryField*** in einem Entwurfsmodell eine konkrete Assoziation zu einer Instanz einer von ***IDLType*** abgeleiteten Metaklasse besitzt. Da die Semantik der Verwendung des *Type-Typed*-Musters an dieser Stelle gerade in der Zuordnung von Signalparametern besteht, ist festzulegen, daß in einem Entwurfsmodell nur Instanzen der Metaklasse ***ValueTypeDef*** erlaubt sind - damit wird die Semantik von ***CarryField*** als ***Typed***-Element präzisiert.

(Constraint 3) In einem Entwurfsmodell darf zu einer ***CarryField***-Instanz nur eine -Instanz der Metaklasse ***ValueTypeDef*** assoziiert werden.

Zur Identifikation der Signalparameter eines Signaltyps muß nun auch noch die Eindeutigkeit der Namen der Signalparameter gefordert werden.

(Constraint 4) In einer Signaltypdefinition müssen die Namen der in der Liste ***values*** enthaltenen Instanzen der Metaklasse ***CarryField*** paarweise verschieden sein.

Die Metamodell-Definition des Konzeptes Signaltyp ist in Abb. 9 illustriert. Die Definition entspricht der im CORBA-IDL-Metamodell enthaltenen Definition des CORBA-IDL-Datentypkonzeptes ***struct*** (vgl.

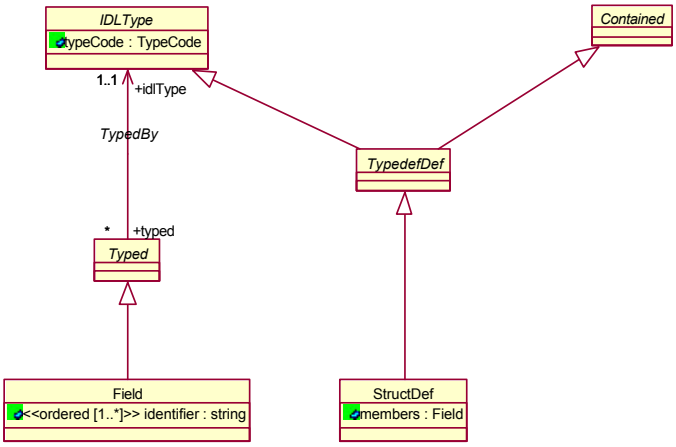


Abb. 10 CORBA-IDL-Metamodell für *StructDef*-Definition

Abb. 10), mit dem Unterschied, daß die Ableitung der Metaklasse ***StructDef*** von der Metaklasse ***Contained*** indirekt über die Metaklasse ***TypedefDef*** erfolgt. Zudem ist die Semantik der Metaklasse ***Field*** als Ableitung von ***Typed*** nicht analog zu *Constraint 3* eingeschränkt.

In [CoRE I], Kapitel3 wurde bereits motiviert, daß die konzeptionelle Trennung von Signaltypen und Signalparametern vorgenommen wurde, um in konkreten Signaltypdefinitionen in einem Entwurfsmodell Eigenschaften der Signaltypdefinition modellieren zu können, die unabhängig von den Signalparametern sind. Im Metamodell müssen zu diesem Zweck die notwendigen Mechanismen bereitgestellt werden. Dabei ist zu beachten, daß die Beschreibung der einem Signaltyp zuzuordnenden Eigenschaften unabhängig von einer konkreten *Component-Support*-Plattform sein soll, damit sich das im Rahmen von *CORE* definierte Metamodell für verschiedene derartige Plattformen wiederverwenden läßt.

Zur Modellierung der Eigenschaften von Signaltypen wird im Metamodell die Metaklasse ***Property*** eingeführt. Diese besitzt ein Attribut ***property_name*** vom Typ ***string*** zur Identifikation der ***Property***-Definition. Weiterhin wird die Metaklasse ***Property*** als Spezialisierung der abstrakten Metaklasse ***Typed*** definiert. Durch

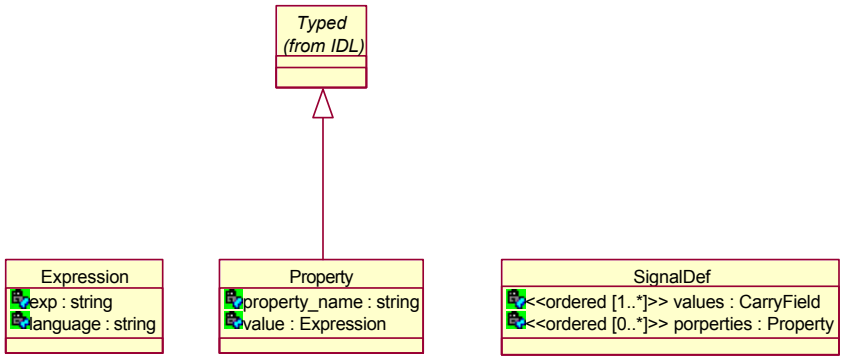


Abb. 11 Metamodell für Eigenschaften von Signaltypen

diese Spezialisierung wird gewährleistet, daß eine konkrete ***Property***-Definition in einem Entwurfsmodell eine Assoziation zu einer Instanz einer Klasse hat, die von der Metaklasse ***IDLType*** abgeleitet ist. Diese Semantik wird nicht weiter eingeschränkt, damit sind als Typen von ***Property***-Definitionen beliebige Typdefinitionen in einem Entwurfsmodell zulässig.

Die Tatsache, daß eine Signaltypdefinition eine Menge von Eigenschaften besitzt, wird im Metamodell durch das Hinzufügen einer weiteren Attributdefinition zu der Metaklasse ***SignalDef*** des Namens ***properties*** modelliert. Der Typ dieser Attributdefinition ist gerade eine Liste (u.U. auch eine leere Liste), deren Listenelemente vom Typ ***Property*** sind. Gefordert werden muß die Eindeutigkeit der ***Property***-Definitionen innerhalb einer Signaltypdefinition.

(Constraint 5) In einer Signaltypdefinition sind die Namen der in der Liste ***properties*** enthaltenen Instanzen von ***Property*** paarweise verschieden.

Auf der Basis der Konzepte Signaltyp und Signalparameter kann nun eine Erweiterung des Konzeptes Interfacetyp um Signalinteraktionen vorgenommen werden.

3.2.4 Erweiterter Interfacetyp

Das Metamodell von CORBA-IDL enthält bereits das Konzept von operationalen Interfacetypen. Im Metamodell ist dieses Konzept durch die Metaklasse ***InterfaceDef*** umgesetzt, die als Ableitung der abstrakten Klasse ***Container*** definiert wurde, um Operationsdefinitionen aufnehmen zu können. Operationsdefinitionen sind Instanzen der Metaklasse ***OperationDef***, die zum Zweck der Aufnahme von konkreten Operationen in Interfacetypen in von der abstrakten Klasse ***Contained*** abgeleitet wurde. Zur Erweiterung des Interfacetypkonzeptes um Signalinteraktion bietet sich die Wiederverwendung des *Container-Contained*-Musters an. Da aber die Semantik der Metaklasse ***InterfaceDef*** aus dem CORBA-IDL-Metamodell unverändert belassen werden soll, muß hier zunächst eine Erweiterung für die Umsetzung des Konzeptes Interfacetyp vorgenommen werden. Dies geschieht durch Einführung der Metaklasse ***EnhancedInterfaceDef*** als Spezialisierung der Metaklasse ***InterfaceDef***. Durch die Spezialisierungsrelation sind die Instanzen der Metaklasse ***EnhancedInterfaceDef*** bereits in der Lage, Operationsdefinitionen und Attributdefinitionen, sowie Datentyp- und Ausnahemendefinitionen zu beinhalten, sowie in einer Generalisierungsbeziehung stehen. Das Metamodell für erweiterte Interfacetypen ist in Abb. 12 dargestellt.

Es wird im folgenden spezifiziert, daß Definitionen erweiterter Interfacetypen in einem Entwurfsmodell nur innerhalb von Instanzen der Klasse ***NamespaceDef*** vorgenommen werden dürfen.

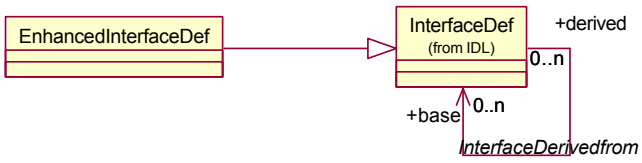


Abb. 12 Metamodell für Erweiterter Interfacetyp

(Constraint 6) Für alle Instanzen der Metaklasse ***EnhancedInterfaceDef*** in einem Entwurfsmodell, für die eine ***Contains***-Assoziation definiert wurde, muß diese ***Contains***-Assoziation zu einer Instanz der Metaklasse ***NamespaceDef*** führen.

Da die Möglichkeit der Definition von Generalisierungsrelationen zwischen Instanzen der Metaklasse ***InterfaceDef*** bereits definiert wurde (durch die Assoziation ***InterfaceDerivedFrom*** im Metamodell), gilt diese Möglichkeit auch für Instanzen der Metaklasse ***EnhancedInterfaceDef***. Instanzen der Metaklasse ***EnhancedInterfaceDef*** können Instanzen der Metaklasse ***InterfaceDef*** spezialisieren. Damit ist eine Wiederverwendung und Spezialisierung von Entwurfsmodellen möglich, die auf dem CORBA-IDL-Metamodell basieren.

Umgekehrt muß jedoch eingeschränkt werden, daß Instanzen der Metaklasse ***InterfaceDef*** keine Instanzen der Metaklasse ***EnhancedInterfaceDef*** spezialisieren können. In einem solchen Fall wäre die Semantik der dann möglicherweise zusätzlich vorhandenen Signal- und *Continous-Media*-Interaktionselmente in der Instanz der Metaklasse ***InterfaceDef*** undefiniert.

(Constraint 7) Für Instanzen der Metaklasse ***EnhancedInterfaceDef*** dürfen Assoziationen im Entwurfsmodell, die auf ***InterfaceDerivedFrom*** basieren, nur von Instanzen der Metaklasse ***EnhancedInterfaceDef*** ausgehen.

3.2.5 Interaktionselement, Consume- und Produce-Definition

Die oben eingeführte Metaklasse ***EnhancedInterfaceDef*** besitzt zunächst nur die Eigenschaften der Metaklasse ***InterfaceDef***, d.h. es ist nur die Angabe von operationalen Interaktionselementen in einem Entwurfsmodell für Instanzen dieser Klasse erklärt. Da mit der Einführung des Konzeptes Signaltyp alle Voraussetzungen für die Einführung von Signalinteraktionselementen an Interfacetypen gegeben sind, kann die Metaklasse ***EnhancedInterfaceDef*** um die entsprechenden Eigenschaften erweitert werden.

Analog zu Operationen im Kontext von operationalen Interfacetypen kann die Definition von Signalinteraktionselementen wiederum durch Anwendung der Modellierungsmuster *Container-Contained* und *Type-Typed* erreicht werden. Ziel der Modellierung ist die Erweiterung der Metaklasse ***EnhancedInterfaceDef*** derart, daß in einem Entwurfsmodell das potentielle Senden bzw. Empfangen von Signalen im Kontext eines Interfacetyps erfaßt werden kann. Potentielles Empfangen/Senden bedeutet, daß über Instanzen eines im Entwurfsmodell beschriebenen Interfacetyps zur Ausführungszeit des verteilten Softwaresystems ausschließlich Instanzen derjenigen Signaltypen versendet bzw. empfangen werden können, für die diese potentielle Möglichkeit bereits im Entwurfsmodell spezifiziert wurde.

Zur Modellierung des potentiellen Empfangs von Signalen im Kontext eines Interfacetyps wird das Konzept *Consume*-Definition in *CORE$_{CEPT}$* aufgenommen. Analog wird das Konzept *Produce*-Definition für die Modellierung des potentiellen Versendens eines Signals im Kontext eines Interfacetyps eingeführt. Im Metamodell sind beide Konzepte durch jeweils eine Metaklasse modelliert (vgl. Abb. 13). Diese Metaklassen ***ConsumeDef*** bzw. ***ProduceDef*** sind von der ebenfalls neu eingeführten abstrakten Metaklasse ***InteractionElement*** abgeleitet, die wiederum die abstrakte Metaklasse ***Contained*** spezialisiert. Die Metaklasse ***InteractionElement*** wird definiert, um gemeinsame Eigenschaften aller Interaktionselemente im Kontext von Interfacetypen kompakt

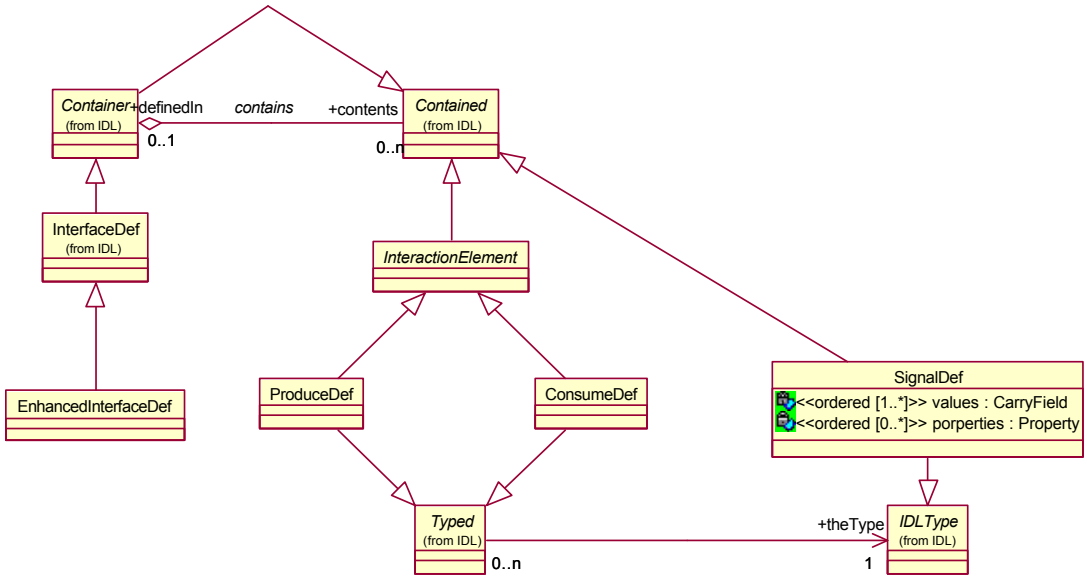


Abb. 13 Metamodell für Signalinterkation

modellieren zu können. In Entwurfsmodellen wird gefordert, daß für Instanzen der von ***InteractionElement*** abgeleiteten Metaklassen die Instanz der von ***Container*** abgeleiteten Metaklasse, in der sie definiert sind, vom Typ ***EnhancedInterfaceDef*** ist (Interaktionselemente dürfen nur in Interfacetypen definiert werden).

(Constraint 8) Für alle Instanzen der von ***InteractionElement*** abgeleiteten Metaklassen in Entwurfsmodellen gilt, daß die ***contains***-Assoziation vorhanden sein muß und zu einer Instanz der Metaklasse ***EnhancedInterfaceDef*** führt.

Die Konzepte *Consume*- und *Produce*-Definition bedeuten den potentiellen Empfang bzw. das potentielle Senden eines Signals - der Bezug zu einer Signaltypdefinition im Entwurfsmodell wird im Metamodell durch die Einführung einer Spezialisierungsrelation der Metaklassen ***ConsumeDef*** bzw. ***ProduceDef*** zu der abstrakten Metaklasse ***Typed*** erreicht. Die einer Instanz von ***ProduceDef*** bzw. ***ConsumeDef*** in einem Entwurfsmodell zugeordnete Instanz muß dann vom Meta-Typ ***SignalDef*** sein.

(Constraint 9) Für alle Instanzen von ***ConsumeDef*** bzw. ***ProduceDef*** in einem Entwurfsmodell muß die über die ***IDLType-Typed***-Assoziation zugeordnete Instanz vom Typ ***SignalDef*** sein.

Die abstrakte Metaklasse ***InteractionElement*** dient zur homogenen Behandlung aller Interaktionsarten im Kontext eines Interfacetyps in einem Entwurfsmodell. Ein Interfacetyp ist Container für Spezialisierungen von ***InteractionElement***, unabhängig von der jeweiligen Interaktionsart. Eine wesentliche Anforderung an *CORE* - die homogene Unterstützung aller Interaktionsarten in Entwurfsmodellen (vgl. [CoRE I], Abschnitt 1.5) - ist damit konzeptionell erreicht. Allerdings sind die operationalen Interaktionselemente Operation und Attribut aus dem CORBA-IDL-Metamodell übernommen und die entsprechenden Metaklassen dort nicht als Spezialisierungen von ***InteractionElement*** definiert. Der Grund ist, daß operationale Interaktionselemente die einzigen von CORBA-IDL unterstützten Interaktionselemente sind. Um die erwähnte Homogenität im Metamodell von *CORE* zu erreichen und gleichzeitig das Metamodell von CORBA-IDL nicht zu verändern, werden zusätzliche Meta-klassen ***OperationDefAsIE*** und ***AttributeDefAsIE*** eingeführt (vgl. Abb. 14). Diese Metaklassen sind sowohl Spezialisierungen der Metaklasse ***InteractionElement*** als auch der Metaklassen ***OperationDef*** bzw. ***AttributDef***. Sie besitzen damit alle Eigenschaften der Konzepte Operation bzw. Attribut

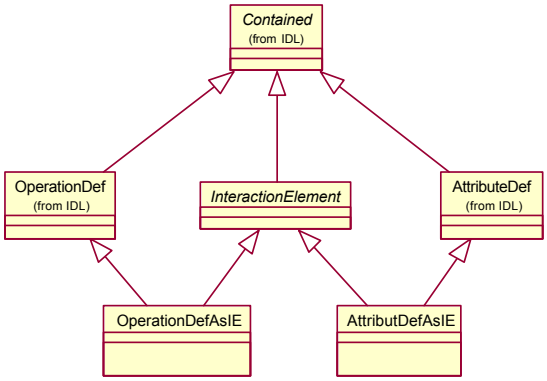


Abb. 14 Metamodell für operationale Interaktionselemente

aus dem CORBA-IDL-Metamodell, sind aber - ananlog zu ***Produce*** und ***Consume*** - Interaktionselemente. Regeln, die für Interaktionselemente definiert wurden, gelten nunmehr auch für ***OperationDefAsIE*** und ***AttributDefAsIE***. Es wird zusätzlich gefordert, daß im Kontext eines Interfacetyps, der basierend auf der Metaklasse ***EnhancedInterfaceDef*** in einem Entwurfsmodell spezifiziert wird, Operationen und Attribute nur unter Benutzung der Metaklassen ***OperationDefAsIE*** und ***AttributDefAsIE*** definiert werden dürfen.

(Constraint 10) Für alle Instanzen von ***EnhancedInterfaceDef*** in einem Entwurfsmodell gilt, daß die Relation ***contains*** nicht zu ***Contained***-Instanzen führen darf, deren spezialisiertester Typ ***OperationDef*** bzw. ***AttributDef*** ist.

3.2.6 Medientyp, Medium und Medienmenge

Die bereits eingeführten Interaktionselemente Operation, Attribut, *Produce* und *Consume* gestatten die Modellierung von operationalen und signalbasierten Interaktionselmenten im Kontext eines Interfacetyps in einem Entwurfsmodell. Eine essentielle Anforderung an einen Konzeptraum, der zur Modellierung von Softwaresystemen im Telekommunikationskontext eingesetzt werden soll, ist zusätzlich eine Unterstützung zur Erfassung von *Continous-Media*-Interaktionselementen (vgl. [CoRE I], Abschnitt 1.5) in Entwurfsmodellen. Analog zu der bereits eingeführten operationalen und signalbasierten Interaktion sind auch hier zunächst die elementaren Bestandteile dieser Interaktionsart im Metamodell zu definieren, bevor die Definition der Interaktionselemente vorgenommen werden kann.

Zweck der *Continous-Media*-Interaktion ist die unidirektionale Kommunikation kontinuierlicher Datenströme von einer Quelle zu einer Senke. Zur Modellierung eines solchen Datenstromes wird in *CoRE* das Konzept Medium eingeführt. Ein und dasselbe Medium kann auf unterschiedliche Art und Weise übertragen werden, verschiedene Standards oder Industrieformate bilden die Grundlage für derartige Übertragungen in Form von Codierungs- und Decodierungsregeln sowie Übertragungsformaten. In einem Entwurfsmodell muß erfaßt werden, welche dieser Formate für die Übertragung eines Mediums geeignet sind. Dies geschieht durch das Konzept Medientyp. Ein und dasselbe Medium kann dabei durch verschiedene Medientypen realisiert (d.h. übertragen) werden.

Im Metamodell werden die Konzepte Medium und Mediatype als Metaklassen ***MediumDef*** und ***MediatypeDef*** eingeführt, die als Spezialisierungen der Metaklasse ***Contained*** definiert sind. Die *Realize*-Relation zwischen Medien und Medientypen wird durch die Definition einer entsprechenden Assoziation zwischen den Metaklassen ***MediumDef*** und ***MediatypeDef*** im Metamodell erfaßt. Dabei wird festgelegt, daß jede Instanz von ***MediumDef*** in einem Entwurfsmodell durch eine oder mehrere Instanzen von ***MediatypeDef*** realisiert wird und umgekehrt jede Instanz von ***MediatypeDef*** mehrere Instanzen von ***MediumDef*** realisieren kann.

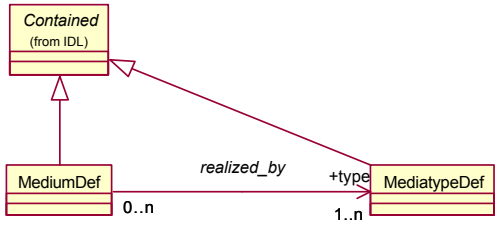


Abb. 15 Metamodell für Medium und Medientyp

Durch eine Regel wird zudem festgelegt, daß die Definitionen von Medien und Medientypen in Entwurfsmodellen nur innerhalb von Namensräumen erfolgen können.

(Constraint 11) Für alle Instanzen von ***MediumDef*** und ***MediatypeDef*** in Entwurfsmodellen gilt: Falls eine Relation ***contains*** definiert ist, muß die entsprechende ***Container***-Instanz vom Typ der Metaklasse ***NamespaceDef*** sein.

In realen Szenarien der *Continous-Media*-Interaktion ist es häufig der Fall, daß nicht ein einziger, atomarer Datenstrom zu übertragen ist, sondern eine Kombination von zwei oder mehreren solcher Datenströme. Dies ist z.B. dann gegeben, wenn Bild- und Tondaten im Rahmen einer Fernsehübertragung gemeinsam gesendet bzw. empfangen werden sollen. Zum Zweck der Zusammenfassung mehrerer Medien wird das Konzept Medienmenge eingeführt. Eine Medienmenge aggregiert dabei ein oder mehrere Medien. Durch die explizite Aufnahme dieses Konzeptes in das Metamodell ist es dann möglich, Eigenschaften, die für alle aggregierten Medien einer Medienmenge gelten sollen, in einem Entwurfsmodell zu beschreiben. Beispiel dafür sind Synchronisationseigenschaften.

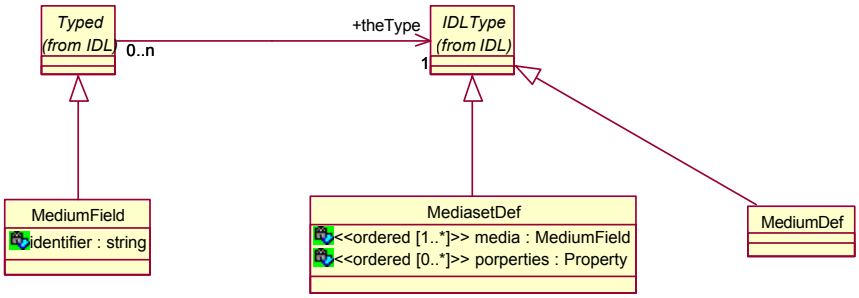


Abb. 16 Metamodell für Medienmenge

Im Metamodell wird zur Reflexion des Konzeptes Medienmenge eine neue Metaklasse ***MediasetDef*** eingeführt. Diese Metaklasse besitzt ein Attribut, dessen Typ als eine nichtleere, geordnete Liste von identifizierbaren Medien definiert ist. Jedes Listenelement hat den Typ ***MediumField***, dieser Typ ist ebenfalls eine neu in das Metamodell aufgenommene Metaklasse mit einem Attribut ***Identifier*** vom Typ ***string***. Über dieses Attribut wird die geforderte Identifikation der Medien im Kontext einer Medienmenge gewährleistet.

(Constraint 12) Für alle Instanzen von ***MediaSetDef*** sind die Werte des Attributes ***identifier*** der in der Liste ***media*** enthaltenen Instanzen von ***MediumField*** paarweise verschieden.

Die Metaklasse ***MediumField*** ist von der abstrakten Metaklasse ***Typed*** abgeleitet. Damit ist über das Modellierungsmuster *Type-Typed* eine Relation zu einer von der abstrakten Klasse ***IDLType*** abgeleiteten Klasse impli-

ziert. Es wird gefordert, daß Instanzen von ***MediumField*** in einem Entwurfsmodell gerade eine Relation zu einer Instanz von ***MediumDef*** haben.

(Constraint 13) Für alle Instanzen von ***MediumField*** in einem Entwurfsmodell führt die konkrete Assoziation zwischen ***Typed*** und ***IDLType*** zu einer Instanz von ***MediumDef***.

Die Metaklasse ***MediaSetDef*** wird hier - wie Datentypen bzw. Signaltypen als analoge Bestanteile der operationalen- und Signalinteraktionselemente - als Spezialisierung der abstrakten Klasse ***IDLType*** definiert. Diese Spezialisierung vereinfacht die nachfolgend präsentierte Einführung der Interaktionselemente für die *Continous-Media*-Interaktion in das Metamodell von *CORE*. Das Konzept Medienmenge gestattet die Erfassung von gemeinsamen Eigenschaften für alle in einer Medienmenge enthaltenen Medien. Da solche Eigenschaften (analog zu Eigenschaften von Signaltypen) von der beabsichtigten Ausführungsumgebung abhängen können, wird für die Modellierung im Rahmen des Metamodells wiederum der schon für Signaltyp gewählte generische Ansatz verwendet. Im Metamodell erhält die Metaklasse ***MediaSetDef*** ein Attribut, dessen Typ eine Liste von ***Property***-Definitionen ist (vgl. Abb.16).

3.2.7 Sink- und Source-Definition

Zur Aufnahme der *Continuous-Media*-Interaktion in *CORE$_{CEPT}$* wird die Definition der Metaklasse ***EnhancedInterfaceDef*** um Assoziationen zu geeigneten Interaktionselementen erweitert. Diese Interaktionselemente sind *Source* (Quelle) und *Sink* (Senke) für die unidirektionale Kommunikation atomarer, kontinuierlicher Datenströme. Ein Interfacetyp kann dabei Quelle bzw. Senke für eine beliebige, endliche Menge von Datenströmen sein. Eine Interfacetypdefinition in einem Entwurfsmodell kann demzufolge eine Menge von *Sink*- bzw. *Source*-Definitionen beinhalten, die aber jeweils im Kontext des Interfacetyps identifizierbar sein müssen. Jede *Source*- bzw. *Sink*-Definition bezieht sich auf genau eine Medienmenge, die die Menge der Medien (also der Datenströme) beschreibt, die zur Ausführungszeit potentiell versendet (*Source*) bzw. empfangen (*Sink*) werden können. Eine *Source*- bzw. *Sink*-Definition an einer Interfacetypdefinition ist eine Strukturinformation, sie bedeutet die Modellierung der Möglichkeit des Empfangens bzw. Versendens von solchen Datenströmen, nicht aber das tatsächliche Zustandekommen einer derartigen Interaktion zur Ausführungszeit.

Im Metamodell werden die Konzepte *Source*- und *Sink*-Definition als neue Metaklassen ***SourceDef*** und ***SinkDef*** erfaßt. Diese Metaklassen sind von der abstrakten Metaklasse ***Interactionelement*** abgeleitet und dürfen somit im Kontext von Instanzen von ***EnhancedInterfaceDef*** in Entwurfsmodellen als Interaktionselemente verwendet werden. (*Container-Contained*-Modellierungsmuster). Damit ist die geforderte Möglichkeit der Identifikation automatisch sichergestellt (vgl. Abb. 17).

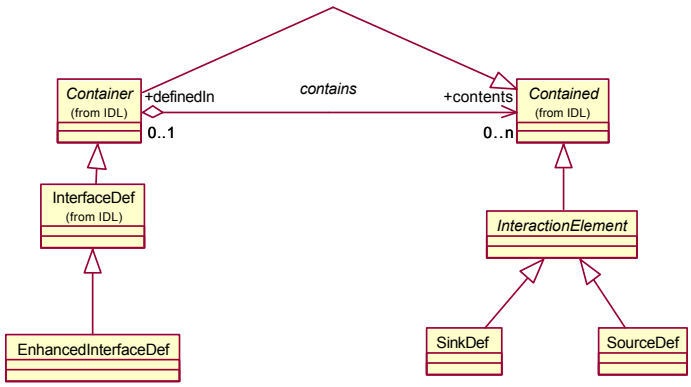


Abb. 17 Metamodell für *Sink* und *Source* als Interaktionselemente

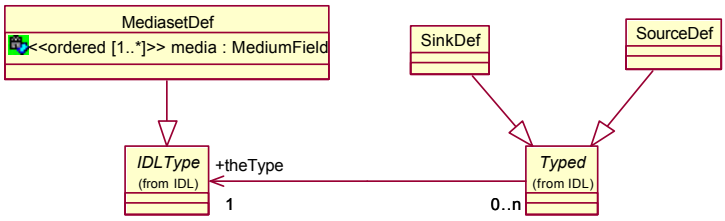


Abb. 18 Assoziation zwischen *SinkDef*, *SourceDef* und *MediasetDef*

Die Relation zu einer zugehörigen Medienmenge wird durch Anwendung des Modellierungsmusters *Type-Typed* erreicht. Die Metaklassen ***SinkDef*** und ***SourceDef*** sind von der abstrakten Metaklasse ***Typed*** abgeleitet und die Metaklasse ***MediasetDef*** von der abstrakten Metaklasse ***IDLType*** (vgl. Abb.18).

Zusätzlich wird gefordert, daß die Assoziation ***IDLType-Typed*** im Entwurfsmodell für Instanzen von ***SourceDef*** bzw. ***SinkDef*** ausschließlich zu Instanzen von ***MediasetDef*** führen darf.

(Constraint 14) Für alle Instanzen von ***SinkDef*** und ***SourceDef*** in einem Entwurfsmodell muß die Assoziation ***IDLType-Typed*** immer zu einer Instanz von ***MediasetDef*** führen.

3.2.8 Diskussion

Die Konzepte Namensraum, Datentyp, Interfacetyp, Operation, Attribut und Ausnahme sind aus CORBA-IDL übernommen und stellen die Basis für die Modellierung der operationalen Interaktionsart dar. Signaltyp, Signalparameter, *Consume*- und *Produce*-Definition dienen der Modellierung der Signalinteraktionsart. Mit der Einführung der Konzepte *Sink*- und *Source*-Definition als Interaktionselemente und der Erweiterung des Konzeptes Interfacetyp ist die Integration aller Interaktionsarten im Kontext eines Interfacetyps im Metamodell gelungen. Eine wichtige Forderung aus [CoRE I], Kapitel1 - die Unterstützung von *Continuous-Media*-Interaktion und die Herstellung des Kontextes zu anderen Interaktionsarten - ist erfüllt. Zentraler Aspekt eines Interfacetyps ist die Menge von an diesem Interfacetyp definierten, eindeutig identifizierbaren Interaktionselementen - unabhängig von der Interaktionsart. Spezifizierte Interaktionselemente sind die Voraussetzung für tatsächliche Interaktionen zur Ausführungszeit, der gemeinsame Kontext ist durch die Interfacetypdefinition gegeben und muß von einer *Component-Support*-Plattform hergestellt werden. Dazu werden geeignete Ableitungsregeln für den entsprechenden Zielcode benötigt. Durch die Definition von $CORE_{WARE}$ in Verbindung mit $CORE_{MAP}$ wird gezeigt, daß sich die konzeptionelle Integration aller Interaktionsarten im Kontext eines Interfacetyps technologisch realisieren läßt.

3.2.9 CO-Typ, Supports- und Requires-Relation

CO-Typ bildet neben Interfacetyp das Basiskonzept der Struktursicht. Die funktionale Dekomposition eines verteilten Softwaresystems während der Erstellung eines Entwurfsmodells resultiert in einer Menge von CO-Typen als Instanzen des Konzeptes CO-Typ. COs als Instanzen der CO-Typen, die bei der Ausführung des verteilten Softwaresystems entstehen, besitzen Identität, Zustand und Verhalten. Sie interagieren zur Ausführungszeit über wohldefinierte Interfaces, deren Typen durch Instanziierung des Konzeptes Interfacetyp ebenfalls im Entwurfsmodell erfaßt sind.

Im Metamodell wird das Konzept CO-Typ durch eine Metaklasse ***COTypeDef*** repräsentiert. Diese Metaklasse ist von der Metaklasse ***InterfaceDef*** (definiert im CORBA-IDL-Metamodell) abgeleitet. Damit besitzen CO-Typen zunächst alle Eigenschaften von Interfacetypen, insbesondere die Möglichkeit, Operations- und

Attributdefinitionen zu beinhalten. Diese Eigenschaft wird ausgenutzt, um den initialen Zugriff auf ein CO nach seiner Erzeugung durch Operationsrufe sowie seine Konfiguration durch das Setzen von Attributwerten zu ermöglichen. Da durch die im Metamodell festgelegte Generalisierung mit der Instanziierung eines CO-Typs ebenfalls ein Interfacetyp instanziiert wird, kann der initiale Zugriff gerade über dieses Interface erfolgen. Im Entwurfsmodell wird festgelegt, welche Operationen zum Zugriff benutzt werden können.

Um die Verwendung der Instanzen der Metaklasse ***COTypeDef*** in Entwurfsmodellen einzuschränken, werden *Constraint*-Definitionen im Metamodell vorgenommen. Es ist die Semantik von CO-Typdefinitionen als Spezialisierung von ***Contained*** zu präzisieren.

(Constraint 15) Die einzigen Modellelemente, in denen konkrete CO-Typen enthalten sein dürfen, sind Instanzen der Metaklasse ***NamespaceDef***.

(Constraint 16) Generalisierungsrelationen zwischen Interfacetypen dürfen keine CO-Typen beinhalten.

In einem Entwurfsmodell ist nun weiterhin zu erfassen, über welche zusätzlichen Interfaces ein CO potentiell an Interaktionen beteiligt sein darf. Dies wird modelliert durch Relationen zwischen dem zugehörigen CO-Typ und denjenigen Interfacetypen, über deren Instanzen Interaktionen ausgeführt werden sollen. Es wird dabei zusätzlich erfaßt, welche Rolle (Klient oder Server) ein CO bezüglich des Interfaces bei einer potentiellen Interaktion einnimmt. Für die Serverrolle wird die Relation *supports* und für die Klientenrolle die Relation *requires* zwischen CO-Typ und Interfacetyp benutzt.

Im Metamodell sind die Relationen *supports* und *requires* als Assoziationen zwischen den Metaklassen ***COTypeDef*** und ***InterfaceDef*** mit den Namen ***supports*** und ***requires*** spezifiziert. Zusätzlich ist festgelegt, daß ein und derselbe Interfacetyp im Entwurfsmodell mehrere *Supports*- und *Requires*-Relationen zu CO-Typen im Entwurfsmodell haben kann, ebenso wie ein und derselbe CO-Typ im Entwurfsmodell in *Supports*- und *Requires*-Relationen zu mehreren Interfacetypen im Entwurfsmodell stehen kann. Eine unidirektionale Navigationsmöglichkeit besteht von einer CO-Typdefinition im Entwurfsmodell zu allen Interfacetypen, zu denen eine *Supports*- oder *Requires*-Relation definiert ist. Damit wird die Semantik der Metaklasse ***InterfaceDef*** - trotz der Einführung neuer Assoziationen - unverändert belassen. Es wird lediglich die Verwendung von Instanzen von ***InterfaceDef*** in einem erweiterten Kontext definiert, ***InterfaceDef*** wird jedoch nicht verändert (s. Abb. 19).

(Constraint 17) Die Assoziationen ***requires*** und ***supports*** müssen immer zu einer Instanz der Klasse ***InterfaceDef*** im Entwurfsmodell führen, Instanzen der Klasse ***COTypeDef*** sind nicht erlaubt.

Da das Ziel der Verwendung von *CORE$_{CEPT}$* in der objektorientierten Modellierung von verteilten Telekommunikationssoftwaresystemen besteht, ist im Metamodell anzugeben, für welche Konzepte die Anwendung von objektorientierten Paradigmen, insbesondere der Generalisierung in Entwurfsmodellen erlaubt ist. Das Konzept CO-Typ gehört zu denjenigen Strukturkonzepten von *CORE$_{CEPT}$*, für die die Anwendung von Generalisierungsrelationen zwischen konkreten Instanzen in Entwurfsmodellen gestattet werden soll, wobei die Mehrfachvererbung zugelassen ist. Aus diesem Grund wird eine reflexive Assoziation der Metaklasse ***COTypeDef*** zur Repräsentation der Vererbungsrelation eingeführt. Eine Navigationsmöglichkeit besteht von der Spezialisierung einer ***COTypeDef***-Instanz zu dieser Instanz, aber nicht umgekehrt. Damit wird die bekannte Semantik des objektorientierten Paradigmas Generalisierung/Spezialisierung im Metamodell reflektiert. Es ist nicht zulässig, daß eine CO-Typdefinition sich selbst spezialisiert.

(Constraint 18) Für alle Instanzen von ***COTypeDef*** in einem Entwurfsmodell darf die Assoziation ***COTypederivedFrom*** weder direkt noch über eine beliebige Kette zu ein und derselben Instanz führen.

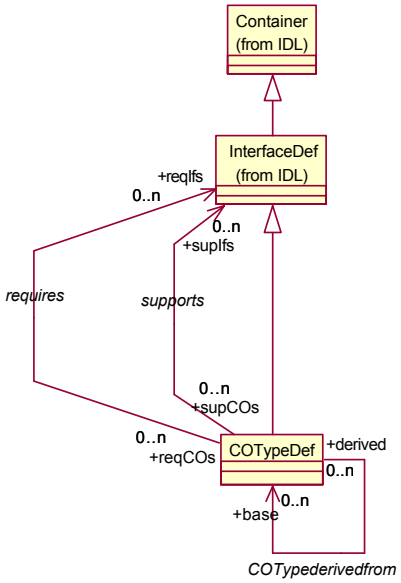


Abb. 19 Metamodell für CO-Typ

3.2.10 Diskussion

Mit dem bisher definierten Metamodell sind alle Konzepte und Relationen der Struktursicht von $CORE_{CEPT}$ erfaßt. Zur Modellierung von Datentypen diente hierbei exemplarisch das Metamodell von CORBA-IDL. Es wurde diskutiert, daß dieses ohne Einfluß auf die auf dem Datentypmodell basierenden Konzepte zu nehmen, austauschbar ist. Diese Eigenschaft ergibt sich aus der konsequenten Anwendung des Modellierungsmusters *Type-Typed*.

Von den in [CoRE I], Kapitel 1 aufgestellten Anforderungen an Entwicklungstechniken für Telekommunikationssoftwaresysteme sind durch die Definition der Konzepte der Struktursicht bereits die Anforderung nach Unterstützung für *Continous-Media*-Interaktion sowie nach Offenheit durch Offenlegung der Schnittstellen konzeptionell erfüllt.

Die vorgestellten strukturellen Aspekte basieren hauptsächlich auf Konzepten und Beziehungen, die bereits aus der Modellierungssprache ITU-ODL bekannt sind. Erweiterungen stellen die Unterstützung *aller* Interaktionsarten (in ITU-ODL nur operationale und *Continuous-Media*-Interaktionen) sowie deren Zusammenfassung im Kontext eines Interfacetyps dar.

3.3 Konfigurationssicht

In der Vergangenheit wurden Telekommunikationssoftwaresysteme oftmals direkt nach der Modellierung von ausschließlich strukturellen Aspekten implementiert [FFH+ 97][BH 98], wenn man von der partiellen Verwendung formaler Beschreibungstechniken, wie z.B. SDL absieht. Das bedeutet, daß die automatische Ableitung von Systemkomponenten (sofern überhaupt realisiert) ausschließlich auf der Basis von strukturellen Informationen über das zu entwickelnde Softwaresystem erfolgte. Alle über die Struktur hinausgehenden Aspekte mußten von den Entwicklern direkt in der jeweiligen Implementierungssprache realisiert werden. Das traf insbesondere für die Konfiguration von Softwaresystemen zu, wobei Konfiguration hierbei die Instanziierung von CO-Typen während der Ausführung von Softwarekomponenten und den Austausch von Interfacereferenzen zwischen diesen Instanzen bedeutet.

Interaktion zwischen den Softwarekomponenten als Basis zur Erfüllung des Zwecks eines verteilten Systems kann nur erreicht werden, wenn der benutzenden Seite (Klient) die zu nutzende Seite (Server) bekannt gemacht wurde, d.h. wenn eine Interfacereferenz auf einen von der Serverseite unterstützten Interfacetyp auf der Klientenseite vorliegt. Erfolgt die automatische Ableitung der Systemkomponenten ausschließlich auf der Basis struktureller Informationen, so ist der Implementierung ein geeigneter Mechanismus zur Konfiguration manuell hinzuzufügen - wobei dann keine Entwurfsmodellinformationen über den genutzten Mechanismus verfügbar sind. In diesem Fall erfolgt die Konfiguration, also der Austausch von Interfacereferenzen auf eine nicht im Entwurfsmodell erfaßte Art und Weise - insbesondere nicht durch automatisch erzeugbare Instruktionssequenzen. Die Anforderung nach flexibler Integration von Softwaresystemkomponenten (vgl. [CoRE I], Abschnitt 1.5) ist dann nicht mehr zu erreichen.

Um die genannte Anforderung zu erfüllen, werden Konzepte für die Konfiguration von verteilten Softwaresystemen explizit in *CoRE$_{CEPT}$* aufgenommen. Grundlage dieses Konfigurationsmechanismus ist die Bereitstellung von definierten Zugriffsmöglichkeiten auf Interfacereferenzen der von CO-Typen unterstützten Interfacetypen. Zur Ausführungszeit können diese Zugriffsmöglichkeiten an Instanzen der CO-Typen genutzt werden. Zusätzlich wird die Definition von Mechanismen zur Hinterlegung von Interfacereferenzen der von CO-Typen benötigten Interfacetypen an diesen CO-Typen ermöglicht.

Die Kombination beider Aspekte, das Beschaffen von Referenzen auf unterstützte Interfacetypen und das Hinterlegen von Referenzen von benötigten Interfacetypen erlaubt die dynamische Konfiguration eines verteilten Softwaresystems zur Ausführungszeit.

Sind diese Informationen Bestandteil eines Entwurfsmodells, so kann bereits auf der Basis des Entwurfsmodells die Bereitstellung eines Konfigurationsmechanismus für ein Softwaresystem erfolgen - ein Eingriff in die Implementierung der Softwarekomponenten ist nicht nötig. Die notwendige programmiersprachliche Unterstützung wird durch die Anwendung von geeigneten Ableitungsregeln automatisch bereitgestellt.

Alle im folgenden beschriebenen Konzepte und Relationen werden im Metamodell innerhalb der ***Package***-Definition ***Configuration*** vorgenommen, da es sich um Konzepte und Relationen der Konfigurationssicht handelt.

3.3.1 Port-, Provided- und Used-Port-Definition

Das Konzept *Port*-Definition dient zum Hinterlegen bzw. Beschaffen von Interfacereferenzen auf von CO-Typen unterstützte bzw. benötigte Interfacetypen. Eine *Port*-Definition muß im Kontext eines CO-Typs eindeutig identifizierbar sein. Auf der Basis dieser Möglichkeit der Identifikation kann zur Ausführungszeit eine geeignete Referenz beschafft bzw. hinterlegt werden.

Dient eine *Port*-Definition zum Beschaffen einer Interfacereferenz, so spricht man von einer *Provided-Port*-Definition. Umgekehrt wird eine *Port*-Definition zum Hinterlegen von Interfacereferenzen als *Used-Port*-Definition bezeichnet.

In realen Softwaresystemen ist es oft erforderlich, mehr als eine Referenz eines Interfacetyps bereitzustellen, bzw. die Hinterlegung von mehr als einer solchen Referenz zu gestatten. Dies ist immer dann der Fall, wenn eine ***1:m*** oder eine ***n:m*** Relation zwischen Klienteninstanzen und Serverinstanzen besteht. Um diese Situation ebenfalls in Entwurfsmodellen geeignet beschreiben zu können, kann eine *Port*-Definition die Auszeichnung ***multiple*** bekommen.

Im Metamodell wird das Konzept *Port*-Definition als abstrakte Metaklasse ***PortDef*** modelliert. Diese Metaklasse ist von der abstrakten Metaklasse ***Contained*** abgeleitet. Die abstrakte Metaklasse ***PortDef*** besitzt zwei Spezialisierungen ***UsePortDef*** und ***ProvidePortDef*** zur Modellierung von *Used-Port*-Definition bzw. *Provided-Port*-Definition (vgl. Abb. 20).

Durch die Verwendung des Modellierungsmusters *Container-Contained* kann auch für *Port*-Definitionen die geforderte eindeutige Möglichkeit der Identifikation gewährleistet werden. Dabei wird festgelegt, daß in Entwurfsmodellen *Port*-Definitionen nur im Kontext von CO-Typdefinitionen vorgenommen werden dürfen.

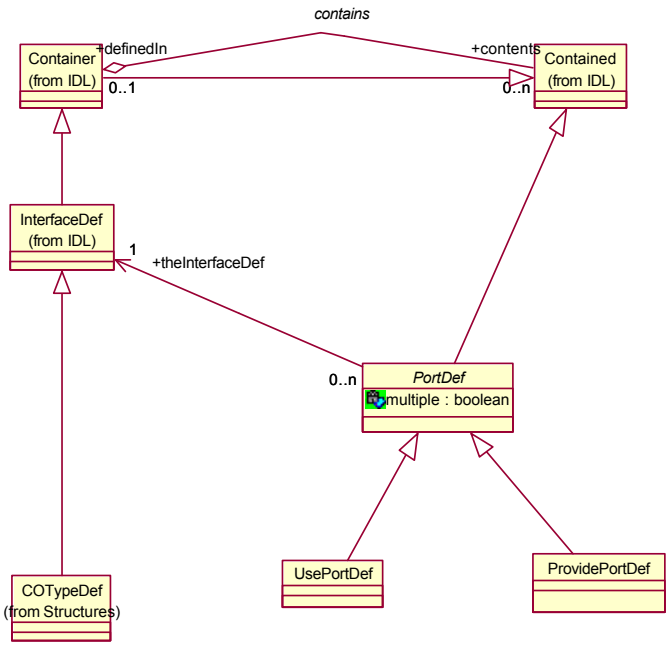


Abb. 20 Metamodell für *Port*

(Constraint 19) Für alle Instanzen von ***UsePortDef*** und ***ProvidePortDef*** in einem Entwurfsmodell gilt, daß die Relation ***contains*** definiert ist und zu einer Instanz von ***COTypedef*** führt.

Eine *Port*-Definition im Kontext eines CO-Typs bezieht sich immer auf einen Interfacetyp, entweder einen von dem CO-Typ unterstützten oder von ihm benötigten Interfacetyp. Diese Relation ist im Metamodell durch die Einführung einer Assoziation zwischen den Metaklassen ***PortDef*** und ***InterfaceDef*** erfaßt. Dabei ist die Zuordnung genau einer Instanz von ***InterfaceDef*** zu einer Instanz von ***PortDef*** zwingend. Umgekehrt können aber mehrere Instanzen von ***PortDef*** derselben Instanz von ***InterfaceDef*** zugeordnet werden. Anschaulich bedeutet dies, daß Referenzen auf ein und denselben Interfacetyp an mehreren unterschiedlichen *Port*-Definitionen im Kontext eines CO-Typs beschafft bzw. hinterlegt werden können.

Da die Klasse ***COTypeDef*** im Metamodell als Spezialisierung der Klasse ***InterfaceDef*** definiert ist, wird zusätzlich gefordert, daß *Port*-Definitionen in Entwurfsmodellen nicht zu CO-Typdefinitionen assoziiert sein dürfen.

(Constraint 20) Für alle Instanzen der Metaklassen, die die Metaklasse ***PortDef*** spezialisieren, gilt, daß die konkrete Assoziation im Entwurfsmodell nicht zu einer Instanz der Klasse ***COTypeDef*** führen darf.

Die Einführung von *Port*-Definitionen in Entwurfsmodellen ist zudem nur dann erlaubt, wenn eine entsprechende *Supports*- bzw. *Requires*-Relation zwischen der Interfacetypdefinition, die zu der *Port*-Definition assoziiert ist und der CO-Typdefinition, die die *Port*-Definition enthält, ebenfalls im Entwurfsmodell vorgenommen wurde.

(Constraint 21) Für alle Instanzen von ***PortDef*** in einem Entwurfsmodell gilt, daß zwischen der durch ***theInterfaceDef*** referenzierten Instanz von ***InterfaceDef*** und der durch ***definedIn*** referenzierten Instanz von ***COTypeDef*** entweder die Assoziation ***requires*** oder ***supports*** bestehen muß. Die Assoziation ***requires*** besteht

genau dann, wenn die Instanz von ***PortDef*** eine Instanz von ***UsePortDef*** ist und die Assoziation ***supports*** genau dann, wenn die Instanz von ***PortDef*** eine Instanz von ***ProvidePortDef*** ist.

Die Kennzeichnung von *Port*-Definitionen mit der Auszeichnung *multiple* erfolgt durch die Einführung eines Attributes an der entsprechenden Metaklasse ***PortDef***. Dieses Attribut trägt den Namen ***multiple*** und ist vom Typ ***boolean***. Es hat in konkreten Entwurfsmodellen den Wert ***true***, wenn die entsprechende *Port*-Definition die Hinterlegung bzw. Beschaffung von mehr als einer Interfacereferenz (des gleichen Interfacetyps) erlauben soll. Andernfalls ist der Wert des Attributs ***false***.

Durch die Einführung des Attributes an der abstrakten Metaklasse ***PortDef*** ist es möglich, die Eigenschaft ***multiple*** sowohl in einer *Used-Port*-Definition als auch in einer *Provided-Port*-Definition zu spezifizieren.

Die Ableitungsregeln für die automatische Ableitung von Softwarekomponenten aus Entwurfsmodellen müssen geeignete Mechanismen bereitstellen, die den Wert des Attributs ***multiple*** für konkrete *Port*-Definitionen berücksichtigen.

Die Konzepte der Konfigurationssicht (*Port*, *Used-Port*, *Provided-Port*) sind unter Benutzung der Konzepte der Struktursicht (Interfacetyp, CO-Typ) definiert. Dadurch ergibt sich die in Abschnitt 3.1 beschriebene Abhängigkeit zwischen den ***Package***-Definitionen ***Structures*** und ***Configuration***.

3.3.2 Diskussion

Die bislang im Metamodell erfaßten Konzepte der Struktur- und Konfigurationssicht erlauben die Erfassung von Informationen in Entwurfsmodellen, die benötigt werden, um die Kommunikation zwischen COs in Softwarekomponenten während ihrer Ausführung zu ermöglichen. Einerseits ist eine vollständige Beschreibung der Interfacetypen mit den Interaktionselementen der verschiedenen Interaktionsarten Bestandteil eines Entwurfsmodells, andererseits ist mit den *Port*-Definitionen beschrieben, welche Möglichkeiten zum Austausch von Interfacereferenzen, also zur Konfiguration bestehen. Eine Softwarekomponente, über deren enthaltene CO-Typen diese Art von Information bekannt ist, kann unter Benutzung einer geeigneten *Component-Support*-Plattform mit anderen Softwarekomponenten integriert, d.h zu einem verteilten Softwaresystem zusammengesetzt werden.

3.4 Implementierungssicht

Die in [CoRE I], Kapitel1 aufgestellten Anforderungen an *CORE* sind unter ausschließlicher Verwendung der Konzepte der Struktur- und Konfigurationssicht nicht zu erfüllen. Insbesondere diejenigen Anforderungen, die Eigenschaften der Implementierung, d.h. der konkreten Codemodule von Softwarekomponenten betreffen, wie:

- flexible Skalierbarkeit,
- flexible Adaptierbarkeit sowie
- kurze Entwicklungszeiten

können mit den bisher im Metamodell erfaßten Konzepten noch nicht durch *CORE* sichergestellt werden. Der Grund besteht darin, daß mit diesen Konzepten in Entwurfsmodellen keinerlei Aussagen darüber getroffen werden können, *wie* das Verhalten von COs an deren Interfaces erbracht werden soll. Durch den Einsatz von Struktur- und Konfigurationskonzepten läßt sich nur modellieren, *welches* Verhalten auf *welche* Art und Weise der Umgebung zugänglich gemacht wird.

Im Kontext von *CORE* wird davon ausgegangen, daß das Verhalten von COs durch das Ausführen von instanziierbaren programmiersprachlichen Konstrukten zur Ausführungszeit erbracht wird. Abstraktionen dieser programmiersprachlichen Konstrukte werden als Artefakte bezeichnet. Um nun eine Zuordnung von Interaktionselementen zu den das Verhalten dieser Interaktionselemente im Kontext von COs realisierenden

Artefakten im Entwurfsmodell vornehmen zu können, wird zunächst das Konzept Artefakt in $CoRE_{CEPT}$ aufgenommen.

Die folgenden Erweiterungen des Metamodells werden in die **Package**-Definition **Implementation** eingeordnet.

3.4.1 Artefakt

Das Verhalten eines CO wird während der Ausführung einer Softwarekomponente durch Instanziierung einer Menge von programmiersprachlichen Konstrukten erbracht, deren Abstraktionen als Artefakte bezeichnet und in $CoRE_{CEPT}$ aufgenommen werden. Konkrete Artefakte werden im Entwurfsmodell CO-Typen zugeordnet. Für die Instanzen (COs) dieser CO-Typen erbringt die Instanziierung und nachfolgende Ausführung der durch die Artefakte modellierten programmiersprachlichen Konstrukte Verhalten, Zustand und Identität. Identische Artefakte können dabei zur Verhaltensrealisierung verschiedener CO-Typen benutzt werden. Artefaktdefinitionen müssen in Entwurfsmodellen eindeutig identifizierbar sein.

Im Metamodell wird das Konzept Artefakt durch die Metaklasse **ArtifactDef** modelliert. Diese Metaklasse ist eine Spezialisierung der abstrakten Metaklasse **Container** und damit auch eine Spezialisierung der abstrakten Metaklasse **Contained**. Damit ist erreicht, daß Artefakte aufgrund der Eigenschaften der abstrakten Klasse **Contained** eindeutig identifizierbar sind. Es wird zusätzlich gefordert, daß Artefaktdefinitionen in Entwurfsmodellen nur innerhalb von Namensraumdefinitionen vorgenommen werden dürfen.

(Constraint 22) Für alle Instanzen von **ArtifactDef** in einem Entwurfsmodell gilt, daß diese Assoziation zu einer Instanz der Metaklasse **NamespaceDef** führt, falls die Assoziation **contains** definiert ist.

Artefakte sind als Abstraktionen von ausführbaren programmiersprachlichen Konstrukten (vgl. [CoRE I], Kapitel 3) definiert. Obwohl mit dieser Definition noch keine Aussage über die Art dieser programmiersprachlichen Konstrukte gemacht wird, repräsentiert bei der Verwendung einer objektorientierten Programmiersprache eine Artefaktdefinition i.allg. das programmiersprachliche Element Klasse. Für Klassen einer objektorientierten Programmiersprache ist die Verwendung von Generalisierung/Spezialisierung möglich - insofern muß hier ebenfalls die Definition von Generalisierungsrelationen zwischen Artefakten als Abstraktionen dieser Klassen erlaubt werden. Dies geschieht im Metamodell durch die Definition einer reflexiven Assoziation für die Metaklasse **ArtifactDef** des Namens **artifactDerivedFrom**. Dabei wird festgelegt, daß eine Artefaktdefinition mehrere andere Artefaktdefinitionen spezialisieren und andererseits eine Generalisierung von mehreren Artefaktdefinitionen sein kann. Eine Spezialisierung einer Artefaktdefinition durch sich selbst ist selbstverständlich nicht erlaubt.

(Constraint 23) Für alle Instanzen von **ArtifactDef** in einem Entwurfsmodell darf die Assoziation **artifactDerivedFrom** weder direkt noch über eine beliebige Kette zu ein und derselben Instanz führen.

3.4.2 Implements-Relation

Die Relation zwischen CO-Typen und denjenigen Artefakten, die das Verhalten an den von ihnen unterstützten bzw. benötigten Interfacetypen realisieren, ist im Metamodell durch die Einführung einer Assoziation **implemented_by** berücksichtigt. Mit dieser Assoziation wird definiert, daß eine CO-Typdefinition durch mehrere Artefakte realisiert werden kann und umgekehrt, daß eine Artefaktdefinition das Verhalten für mehrere CO-Typdefinitionen implementieren kann.

Das Metamodell von **ArtifactDef** und der Assoziationen **implemented_by** und **artifactDerivedFrom** ist in Abb.21 dargestellt.

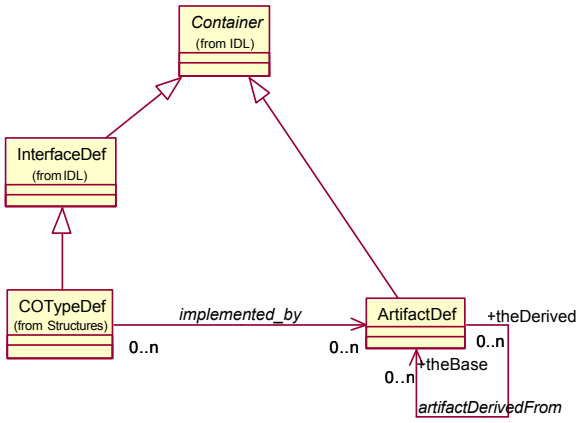


Abb. 21 Metamodell für Artefakt

3.4.3 Implementierungselement

Mit der Metaklasse ***ArtifactDef*** im Metamodell von *CORE* ist nunmehr die Voraussetzung geschaffen, um eine Zuordnung der Interaktionselemente der von einem CO-Typ unterstützen bzw. von ihm benötigten Interfacetypen auf Artefakte vornehmen zu können. Damit ist im Entwurfsmodell festgelegt, welche Artefakte für die Relaisierung eines bestimmten Teils des Verhaltens eines CO-Typs verantwortlich sind.

Die Zuordnung eines Interaktionselements zu einem Artefakt wird in *CORE$_{CEPT}$* als Implementierungselement bezeichnet. Implementierungselemente müssen, um die automatische Ableitung von Softwarekomponenten aus einem Entwurfsmodell zu ermöglichen, im Kontext eines Artefaktes eindeutig identifizierbar sein.

Im Metamodell wird zur Repräsentation des Konzeptes Implementierungselement die Metaklasse ***ImplementationElementDef*** eingeführt. Diese Metaklasse spezialisiert die abstrakte Metaklasse ***Contained***. Damit ist die geforderte eindeutige Möglichkeit der Identifikation im Kontext der definierenden ***Container***-Instanz erreicht. Es wird festgelegt, daß für konkrete Instanzen von ***ImplementationElementDef*** in einem Entwurfsmodell die durch die Assoziation ***contains*** spezifizierte Instanz der von ***Container*** abgeleiteten Metaklasse immer vom Typ ***ArtifactDef*** ist.

(Constraint 24) Für alle Instanzen von ***ImplementationElementDef*** in einem Entwurfsmodell gilt, daß die Assoziation ***contains*** definiert und die entsprechende Instanz vom Typ ***ArtifactDef*** ist.

Zusätzlich wird gefordert, daß Artefaktdefinitionen ausschließlich Container für Implementierungselemente sind.

(Constraint 25) Für alle Instanzen von ***ArtifactDef*** in einem Entwurfsmodell gilt, falls die Assoziation ***contains*** definiert ist und zu einer Instanz einer von ***Contained*** abgeleiteten Metaklasse führt, so ist diese Instanz vom Typ ***ImplementationElementDef***.

Ein Implementierungselement ist immer für die Realisierung des Verhaltens genau eines Interaktionselements verantwortlich. Dabei ist zu unterscheiden, ob das Verhalten des Interaktionselements für einen unterstützten oder einen benötigten Interfacetyp durch das Implementierungselement realisiert werden soll. Nach der Art der Interaktionselemente und dem beabsichtigten Fall (unterstützter Interfacetyp oder benötigter Interfacetyp) ist dabei von dem Implementierungselement das Verhalten entsprechend Tab. 2 zu realisieren.

Interaktionselement	unterstützter Interfacetyp	benötigter Interfacetyp
Operation	Implementierung des Operationsverhaltens	nicht erlaubt
Attribut	Implementierung des Setzens oder Lesens des Attributwertes	nicht erlaubt
Sink	Implementierung des Verhaltens zum Empfangen von kontinuierlichen Datenströmen	Implementierung des Verhaltens zum Senden von kontinuierlichen Datenströmen
Source	Implementierung des Verhaltens zum Senden von kontinuierlichen Datenströmen	Implementierung des Verhaltens zum Empfangen von kontinuierlichen Datenströmen
Produce	Implementierung des Verhaltens beim Senden eines Signals	Implementierung des Verhaltens beim Empfangen eines Signals
Consume	Implementierung des Verhaltens beim Empfangen eines Signals	Implementierung des Verhaltens beim Senden eines Signals

Tab.2 Von Implementierungselementen zu realisierendes Verhalten

Die Relation eines Implementierungselements zu dem von ihm realisierten Interaktionselement wird im Metamodell durch die Assoziation ***realizes*** modelliert. Dabei ist festgelegt, daß jedes Implementierungselement genau ein Interaktionselement realisiert, daß aber umgekehrt jedes Interaktionselement von mehreren Implementierungselementen realisiert werden kann. Ist letzteres der Fall, so ist es Aufgabe der Ableitungsregeln und der *Component-Support*-Plattform, zur Ausführungszeit für ein konkretes Interaktionselement ein geeignetes Implementierungselement zu ermitteln.

Die Verantwortung eines Implementierungselements für die Realisierung des Verhaltens eines Interaktionselements bezüglich eines *unterstützten* oder aber eines *benötigten* Interfacetyps wird im Entwurfsmodell durch die Belegung eines Attributes an Instanzen der Klasse ***ImplementationElementDef*** ausgedrückt. Dieses Attribut wird für die Metaklasse ***ImplementationElementDef*** definiert, erhält den Namen ***Case*** und den Typ ***ImplementationCase***. ***ImplementationCase*** ist eine *Enumeration*-Definition mit den *Enumerator*-Elementen ***use*** und ***supply***. Hat das Attribut in einem konkreten Entwurfsmodell den Wert ***supply***, so implementiert das Implementierungselement das Interaktionselement für einen unterstützten Interfacetyp, hat es den Wert ***use***, so wird das Verhalten des Interaktionselements für einen benötigten Interfacetyp realisiert. Die entsprechend Tab. 2 nicht erlaubten Fälle sind auszuschließen.

(Constraint 26) Für alle Instanzen von ***ImplementationElementDef*** gilt: Falls der Wert des Attributes ***ImplementationCase use*** ist, darf die Instanz der von ***InteractionElement*** abgeleiteten Metaklasse nicht von Typ ***OperationDef*** oder ***AttributDef*** sein.

Zusätzlich zu der obigen *Constraint*-Definition sind noch weitere Forderungen an Entwurfsmodelle zu stellen, die die Konzepte Implementierungselement und Artefakt verwenden. So ist zu fordern, daß das von einem konkreten Implementierungselement realisierte Interaktionselement an einem Interfacetyp definiert

ist, der von einem CO-Typ benötigt oder bereitgestellt wird. Dieser CO-Typ muß durch das Artefakt implementiert werden, das das Implementierungselement enthält.

(Constraint 27) Für alle Instanzen von ***ImplementationElementDef*** in einem Entwurfsmodell müssen eine Instanz von ***COTypeDef*** und eine Instanz von ***InterfaceDef*** derart existieren, daß:

- die Instanz von ***InteractionElement***, auf das durch die Assoziation ***realizes*** von der Instanz von ***ImplementationElementDef*** verwiesen wird, durch die Assoziation ***contains*** mit der Instanz von ***InterfaceDef*** verbunden ist,
- eine Verbindung zwischen der Instanz von ***InterfaceDef*** durch eine der Assoziationen ***requires*** oder ***supports*** mit der Instanz von ***COTypeDef*** besteht und
- die Instanz von ***COTypeDef*** durch die Assoziation ***implements*** mit der Instanz von ***ArtifactDef*** verbunden ist.

Weiterhin ist die Belegung des Attributes ***Case*** abhängig davon, ob es für das zugeordnete Interaktionselement tatsächlich einen entsprechenden Interfacetyp im Entwurfsmodell gibt, der von einem CO-Typ unterstützt bzw. benötigt wird.

(Constraint 28) Für alle Instanzen von ***ImplementationElementDef*** in einem Entwurfsmodell müssen eine Instanz von ***InterfaceDef*** und eine Instanz von ***COTypeDef*** entsprechend Constraint 27 existieren, wobei zusätzlich gilt:

- ist der Wert des Attributes ***Case use***, so muß die Assoziation ***requires*** zwischen der Instanz von ***COTypeDef*** und der Instanz von ***InterfaceDef*** definiert sein,
- ist der Wert des Attributes ***Case supply***, so muß die Assoziation ***supports*** zwischen der Instanz von ***COTypeDef*** und der Instanz von ***InterfaceDef*** definiert sein.

Das Metamodell für Implementierungselement ist in Abb. 22 dargestellt.

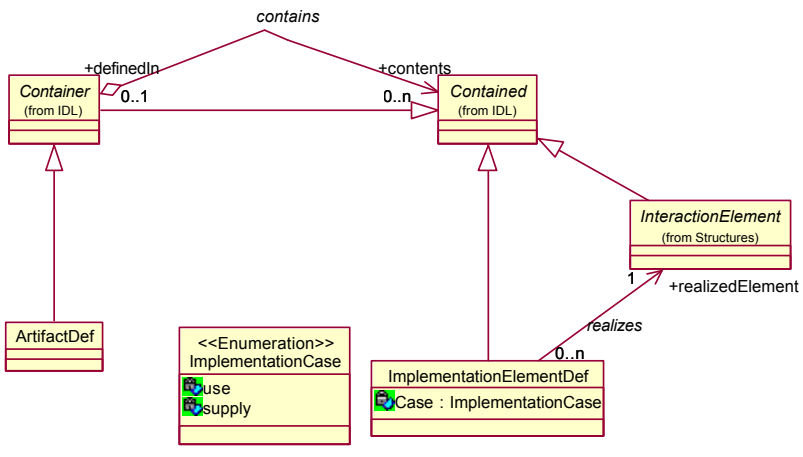


Abb. 22 Metamodell für Implementierungselement

3.4.4 Zustandsattribut

Die Realisierung der Implementierungselemente einer Artefaktdefinition erfordert i.allg. den Zugriff auf Zustandsinformation der entsprechenden CO-Typdefinition. Soll die Verwaltung und Bereitstellung dieser Zustandsinformation durch eine *Component-Support*-Plattform und geeignete Ableitungsregeln automatisch

erfolgen, so muß in einem Entwurfsmodell die Zustandsinformation eines CO-Typs beschrieben werden. Zusätzlich muß erfaßt werden, welcher Teil dieser Zustandsinformation für die Realisierung eines konkreten Implementierungselements relevant ist.

Die Beschreibung der Zustandsinformation erfolgt typbasiert durch die identifizierbare Angabe von Datentypen, deren Instanzen zur Ausführungszeit gerade eine konkrete Zustandsinformation repräsentieren.

Im Metamodell wird das Konzept Zustandsattribut durch eine Metaklasse ***StateDef*** repräsentiert, die von der abstrakten Metaklasse ***Typed*** abgeleitet ist. Durch diese Konstruktion im Metamodell sind als Zustand von CO-Typen alle Spezialisierungen der abstrakten Metaklasse ***IDLType*** zulässig. Hier wurde wiederum das Modellierungsmusters *Type-Typed* eingesetzt. Das Metamodell für Zustandsinformationen ist in Abb. 23 dargestellt.

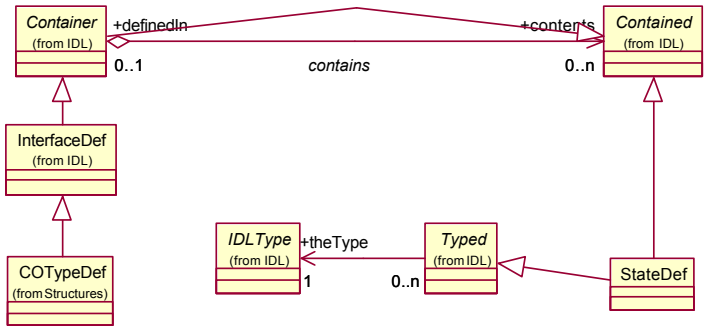


Abb. 23 Metamodell für Zustandsattribut

Es wird gefordert, daß als Typ eines Zustandsattributes nur CORBA-IDL-Datentypen erlaubt sind. Diese Einschränkung ist notwendig, da in dem hier definierten Metamodell Spezialisierungen der Metaklasse ***IDLType*** vorgenommen wurden, deren Verwendung als Zustandsinformation eines CO-Typs nicht sinnvoll ist, z.B. ***SignalDef***[1].

(Constraint 29) Für alle Instanzen der Metaklasse ***StateDef*** in einem Entwurfsmodell gilt, daß die über die Assoziation ***IDLType-Typed*** zugeordnete Instanz der Metaklasse ***IDLType*** einen Typ haben muß, der in der ***Package***-Definition ***IDL*** definiert ist.

Die Metaklasse ***StateDef*** ist indirekt von der abstrakten Metaklasse ***Contained*** abgeleitet, wodurch die Möglichkeit der Identifikation von Zustandsattributen im Kontext des definierenden Containers sichergestellt ist. Da konkrete Zustandsinformationen immer im Kontext eines COs verwaltet werden, wird die entsprechenden Zustands-attribute der CO-Typdefinition zugeordnet.

(Constraint 30) Für alle Instanzen der Metaklasse ***StateDef*** in einem Entwurfsmodell gilt, daß die Assoziation ***contains*** definiert ist und zu einer Instanz der Metaklasse ***COTypeDef*** führt.

Die Zuordnung derjenigen Zustandsattribute, die für die Realisierung eines Implementierungselements relevant sind, erfolgt im Metamodell durch die Definition einer Assoziation ***provided_to*** zwischen den Metaklassen ***ImplementationElementDef*** und ***StateDef***. Es wird festgelegt, daß einer Instanz von ***ImplementationElementDef*** eine Menge von Instanzen von ***StateDef*** zugeordnet werden kann und umgekehrt jede Instanz von ***StateDef*** mehreren Instanzen von ***ImplementationElementDef*** zugeordnet werden kann. Dabei ist gewährleistet, daß im

1. Aktuelle Signalparameter können natürlich Bestandteil der Zustandsinformation eines CO sein (z.B. Parameter des letzten empfangenen Signals), jedoch nicht das Signal selbst.

Entwurfsmodell von einer Instanz von ***StateDef*** zu den zugehörigen Instanzen von ***ImplementationElementDef*** navigiert werden kann.

Durch entsprechende Ableitungsregeln zur automatischen Erzeugung von Softwarekomponenten sowie eine geeignete *Component-Support*-Plattform kann nun die Bereitstellung derjenigen Zustandsinformation eines COs, die für die Implementierung des Verhaltens des Implementierungselements benötigt wird, automatisch zur Ausführungszeit erfolgen. Dies ist dadurch bedingt, daß alle relevanten Zustandsattribute für ein CO im Entwurfsmodell bekannt sind und zudem diejenige Teilmenge dieser Zustandsinformation, die für ein bestimmtes Implementierungselement bereitzustellen sind.

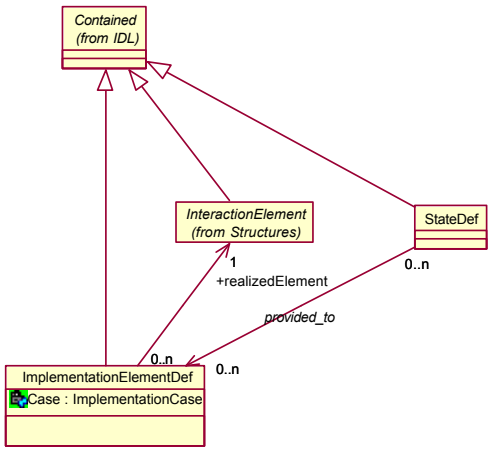


Abb. 24 Metamodell für Implementierungselement mit Zustandsattribut

Ein Entwurfsmodell ist allerdings nur dann korrekt definiert, wenn die für ein Implementierungselement benötigten Zustandsattribute auch im Kontext eines durch das entsprechende Artefakt realisierten CO-Typs definiert sind.

(Constraint 31) Für alle Instanzen von ***ImplementationElementDef*** in einem Entwurfsmodell gilt, daß für alle Instanzen von ***StateDef***, die der Instanz von ***ImplementationElementDef*** zugeordnet sind, eine Instanz von ***COTypeDef*** existiert,

- die Constraint 27 genügt und
- für die die Assoziation ***contains*** zu der Instanz von ***StateDef*** definiert ist.

3.4.5 Diskussion

Mit den bisher eingeführten Konzepten und Relationen der Implementierungssicht (Artefakt, *Implements*-Relation, Zustandsattribut, Implementierungselement) ist es möglich, einem Entwickler bei der Implementierung eines Softwaresystems umfangreiche Unterstützung durch die Modellierung und die anschließende automatische Ableitung von Softwarekomponenten aus dem resultierenden Entwurfsmodell zu gewähren. Die automatische Ableitung von Softwarekomponenten stellt dabei programmiersprachliche Mechanismen bereit, die die Anbindung der Kommunikationsmechanismen der *Component-Support*-Plattform entsprechend den spezifizierten unterstützten und benötigten Interfacetypen von CO-Typen an die ebenfalls im Entwurfsmodell hinterlegte Implementierungsstruktur gewährleisten. Die Aufgabe des Entwicklers besteht nun in der Realisierung des Verhaltens der für die Interaktionselemente zuständigen Implementierungselemente.

Durch die Trennung der Modellierung von Interfacetypen von CO-Typen und der Modellierung der programmiersprachlichen Konstrukte, die deren Verhalten realisieren, ist eine hohe Flexibilität im Entwurf des verteilten Softwaresystems gegeben. Unabhängig von einer (möglicherweise vorgegebenen) Interfacetypdefinition läßt sich eine geeignete Implementierung des Verhaltens auswählen, deren Abstraktion im Entwurfsmodell erfaßt wird. Die Bindung der Interfacetypen mit dieser Implementierung wird im Entwurfsmodell beschrieben und automatisch hergestellt. Liegen bereits Implementierungen für eine bestimmte Funktionalität vor, so können diese benutzt werden.

Es ist somit bereits ein wichtiger Beitrag zur Erfüllung der aufgestellten Forderungen nach flexibler Adaptierbarkeit und kurzen Entwicklungszeiten geleistet. Allerdings ist es mit den bisher eingeführten Beschreibungsmitteln noch nicht möglich, im Entwurfsmodell festzulegen, wie die Instanziierung der Artefakte vorzunehmen ist. Einerseits soll diese Instanziierung wiederum durch automatisch erzeugte programmiersprachliche Mechanismen erfolgen, um den Entwickler von dieser Aufgabe zu entbinden. Andererseits ist eine flexible Gestaltung dieser automatischen Erzeugung notwendig, um weitere Anforderungen, z.B. nach Skalierbarkeit erfüllen zu können.

Ein auch aus anderen Ansätzen (*Policies* in [OMG CORBA]) bekanntes Mittel zur Lösung dieses Konflikts besteht darin, die automatische Erzeugung der programmiersprachlichen Mechanismen geeignet zu parameterisieren, wobei die entsprechenden Parameter in Form von vordefinierten Werten (*Policies*) im Entwurfsmodell erfaßt werden.

3.4.6 Instanziierungsmuster

Das Konzept Instanziierungsmuster dient zur Auswahl einer Strategie, die zur Instanziierung von Artefakten zur Ausführungszeit eingesetzt werden soll. Die Umsetzung dieser Auswahl durch geeignete programmiersprachliche Mechanismen erfolgt durch die automatische Ableitung von Softwarekomponenten aus Entwurfsmodellen.

Die Menge der zur Verfügung stehenden Werte wird im Metamodell durch einen Aufzählungstyp (*Enumeration*) beschrieben. Die definierten Elemente (*Enumerators*) dieses Typs sind:

- ***ARTIFACT_PER_REQUEST*** - für jeden Zugriff auf ein Implementierungselement des Artefakts, d.h. beim Zustandekommen der Interaktion bezüglich des zugordneten Interaktionselements, wird eine neue Instanz des Artefaktes erzeugt,
- ***ARTIFACT_POOL*** - eine gewisse Anzahl von Artefaktinstanzen wird vor dem Zustandekommen von Interaktionen erzeugt, beim Zugriff auf ein zugehöriges Implementierungselement wird eine entsprechende Artefaktinstanz aus dieser Menge ausgewählt,
- ***SINGLETON*** - genau eine Instanz des Artefaktes wird initial erzeugt, jeder Zugriff auf Implementierungselemente des Artefakts wird durch diese Instanz realisiert,
- ***USER_DEFINED*** - der programmiersprachliche Mechanismus zur Erzeugung von Artefakten wird durch den Entwickler bereitgestellt und daher nicht durch die Ableitungsregeln für die automatische Ableitung von Softwarekomponenten berücksichtigt.

Die Metaklasse ***ArtifactDef*** erhält ein zusätzliches Attribut ***instantiation_policy*** vom Typ ***InstantiationPolicy***. In Entwurfsmodellen kann für Artefaktdefinitionen die gewünschte Art ihrer Instanziierung durch die Belegung dieses Attributes vorgegeben werden (vgl. Abb. 25).

Einige Werte, wie z.B. ***ARTIFACT_POOL*** erfordern dabei die zusätzliche Angabe von Parametern. Im Falle von ***ARTIFACT_POOL*** ist dies die Anzahl der initial zu erzeugenden Instanzen. Um die flexible Spezifikation solcher Parameter in Entwurfsmodellen zu erlauben, wurde hier - analog zu Eigenschaften von Signaltypen - das Konzept *Property* verwendet. Damit können Instanzen von ***ArtifactDef*** in Entwurfsmodellen über beliebige Instanzen von ***Property*** und damit beliebige Parameter für die Instanziierung verfügen.

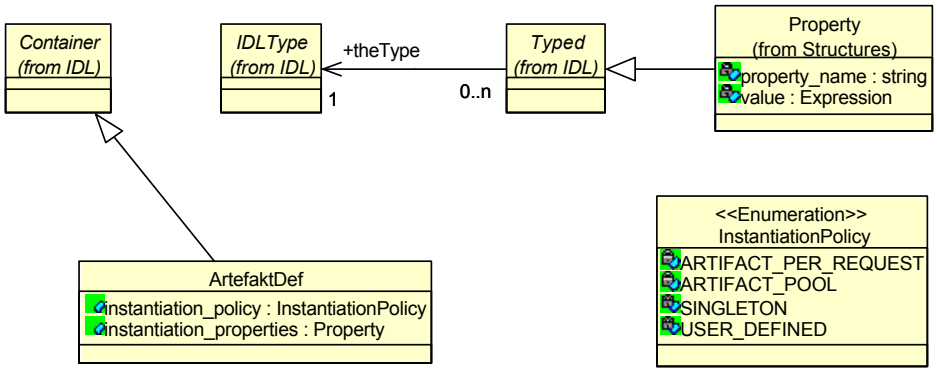


Abb. 25 Metamodell für Instanziierungsmuster

Im Metamodell wird dieser Sachverhalt durch die Einführung eines zusätzlichen Attributes ***inatantiation_properties*** an der Metaklasse ***ArtifactDef*** berücksichtigt. Der Typ dieses Attributes ist eine Liste von Instanzen der Metaklasse ***Property***.

Die eingeführten Konzepte der Implementierungssicht basieren auf Konzepten der Struktursicht. Es ergibt sich die in Abschnitt 3.1 beschriebene Abhängigkeit zwischen den ***Package***-Definitionen ***Structures*** und ***Implementation*** im Metamodell von *CORE*.

3.5 Deployment-Sicht

Die bisher eingeführten Konzepte und Relationen von *CORE$_{CEPT}$* gestatten in Entwurfsmodellen die Beschreibung von CO-Typen sowie deren potentiellen Interaktionsmöglichkeiten über unterstützte und benötigte Interfacetypen. Zusätzlich können die Konfigurationsmöglichkeiten für diese CO-Typen spezifiziert werden. Weiterhin lassen sich durch Artefakte die programmiersprachlichen Konstrukte beschreiben, die das Verhalten, den Zustand und die Identität der COs realisieren. Im Kontext von Artefakten können die Realisierungsbeziehungen zwischen Implementierungselementen und Interaktionselementen von Interfacetypen angegeben werden. Darüber hinaus erfolgt die Erzeugung von Instanzen dieser programmiersprachlichen Konstrukte auf der Grundlage von Regeln, die im Entwurfsmodell definiert sind.

Der nächste Schritt besteht darin, die Softwarekomponenten selbst innerhalb des Entwurfsmodells zu erfassen und diesen zu realisierende CO-Typen zuzuordnen. Ohne diese Zuordnung ist die automatische Ableitung von Softwarekomponenten aus dem Entwurfsmodell nicht möglich, da nicht bekannt ist, welcher Teil des Entwurfsmodells, d.h. welche CO-Typen Bestandteil welcher Softwarekomponenten werden sollen.

Auf der Grundlage der Information über Softwarekomponenten und den von ihnen realisierten CO-Typen kann die Menge der Modellelemente, die für die automatische Ableitung der Softwarekomponenten herangezogen werden muß, durch ein geeignetes Verfahren ermittelt werden. Dabei werden ausgehend von den CO-Typen im Entwurfsmodell alle Assoziationen zu anderen Modellelementen traversiert, diese Modellelemente der Menge der zu realisierenden Modellelemente hinzugefügt, und das Verfahren dann rekursiv auf die neu hinzugewonnenen Elemente angewendet.

3.5.1 Softwarekomponente und Realize-Relation

Eine Softwarekomponente enthält Codemodule, deren Ausführung die Identität, den Zustand und das Verhalten von COs realisieren. Softwarekomponenten werden im Entwurfsmodell durch Instanziierungen des Konzeptes Softwarekomponente modelliert. Wichtigste Information über Softwarekomponenten ist die Angabe derjenigen CO-Typen, die während der Ausführung einer Softwarekomponente instanziiert werden

können - für die also die Softwarekomponente alle zur Erbringung des Verhaltens und zur Realisierung von Zustand und Identität benötigten Codemodule enthält.

Im Metamodell wird das Konzept Softwarekomponente durch die Einführung einer neuen Metaklasse ***ComponentDef*** erfaßt. Die Klasse ***ComponentDef*** ist von der abstrakten Metaklasse ***Container*** und damit auch von ***Contained*** abgeleitet, also im Entwurfsmodell eindeutig identifizierbar. Es wird gefordert, daß Instanzen von ***ComponentDef*** nur innerhalb von Instanzen von ***NamespaceDef*** definiert werden dürfen.

(Constraint 32) Für alle Instanzen von ***ComponentDef*** in einem Entwurfsmodell gilt, falls die Assoziation ***contains*** definiert ist, führt sie zu einer Instanz der Metaklasse ***NamespaceDef***.

Jede Softwarekomponente realisiert eine Menge von CO-Typen. Diese Eigenschaft wird im Metamodell durch die Definition einer Aggregation ***realized_in*** zwischen den Metaklassen ***COTypeDef*** und ***ComponentDef*** widergespiegelt. Dabei wird festgelegt, daß ein und dieselbe Softwarekomponente mehrere CO-Typen realisieren kann und umgekehrt jeder CO-Typ durch eine Menge von Softwarekomponenten implementiert sein darf (vgl. Abb. 26). Zusätzlich ist die Navigation von einer Instanz von ***ComponentDef*** in einem Entwurfsmo-

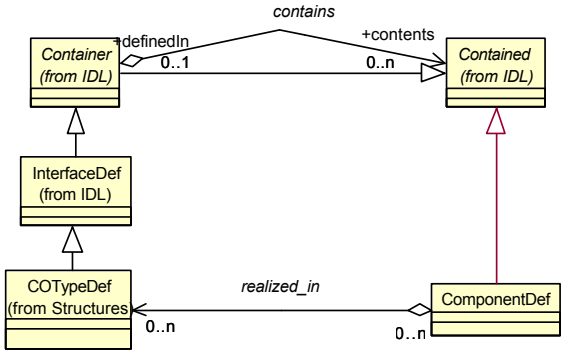


Abb. 26 Metamodell für Softwarekomponente

dell zu allen Instanzen von ***COTypeDef*** möglich, die von der Softwarekomponente implementiert werden. Diese Navigationseigenschaft vereinfacht insbesondere die Ausführung des oben beschriebenen Verfahrens zur Ermittlung derjenigen Modellelemente, die für die automatische Ableitung der Softwarekomponente benötigt werden.

In [CoRE I], Kapitel 1 wurde bereits motiviert, daß Softwarekomponenten in Form einer physikalischen Repräsentation auf geeigneten Maschinen verfügbar gemacht, installiert und anschließend ausgeführt werden. Dabei ist jedoch zu berücksichtigen, daß eine Softwarekomponente über bestimmte Eigenschaften verfügen muß. Diese Eigenschaften sollen in Entwurfsmodellen spezifiziert werden können, um für eine werkzeugunterstützte Verteilung und Installation von Softwarekomponenten nutzbar zu sein.

Eine solche Werkzeugunterstützung geht noch über die bereits in [CoRE I], Abschnitt 1.5 geforderte Unterstützung der Modellierung und der automatischen Ableitung von Softwarekomponenten hinaus. Dabei ist davon auszugehen, daß das Problem der Verteilung der Softwarekomponenten auf einer Menge von Maschinen bei einer hinreichend großen Menge von Softwarekomponenten und Maschinen manuell nicht einfach zu lösen ist, da die Eigenschaften der Softwarekomponenten berücksichtigt werden müssen. Eine Werkzeugunterstützung kann die Eigenschaften von Softwarekomponenten, die in Entwurfsmodellen beschrieben sind, benutzen, um für die Ausführung geeignete Maschinen einer verteilten Ausführungsumgebung zu ermitteln. Die Installation kann dann ebenfalls automatisch auf diesen Maschinen erfolgen.

Die Voraussetzung für eine solche erweiterte Werkzeugunterstützung wird im Metamodell durch die Definition eines Attributes ***properties*** an der Metaklasse ***ComponentDef*** geschaffen. Der Typ dieses Attributes ist eine Liste von identifizierbaren Elementen des Typs ***Property***. Somit lassen sich beliebige Eigenschaften von Softwarekomponenten in Entwurfsmodellen erfassen.

In realen Softwaresystemen erfordern einzelne Softwarekomponenten oft die Präsenz anderer Softwarekomponenten, um ihre Ausführbarkeit zu gewährleisten. Ein Beispiel für eine solche Abhängigkeit wäre die Benutzung einer mathematischen Bibliothek durch die Implementierung einer Softwarekomponente. Soll der *Deployment*-Vorgang wie oben beschrieben automatisch ausgeführt werden, so ist auch diese Information bereitzustellen. Im Metamodell werden Abhängigkeiten zwischen Softwarekomponenten durch die Definition einer neuen Metaklasse ***ComponentDependencyDef*** berücksichtigt. Instanzen dieser Metaklasse stellen gerade die Abhängigkeit zwischen zwei Softwarekomponenten im Entwurfsmodell dar. Damit die beiden beteiligten Softwarekomponenten für eine solche Abhängigkeit identifizierbar sind, werden im Metamodell zusätzliche Assoziationen eingeführt.

Die Assoziation ***dependencies*** ist als Aggregation zwischen den Metaklassen ***ComponentDef*** und ***ComponentDependencyDef*** definiert. Es ist festgelegt, daß jeder Instanz von ***ComponentDef*** mehrere Instanzen von ***ComponentDependencyDef*** zugeordnet werden können und umgekehrt jeder Instanz von ***ComponentDependencyDef*** genau eine Instanz von ***ComponentDef*** zugeordnet ist. Die Assoziation ***dependend_from*** ist zwischen den Metaklassen ***ComponentDependencyDef*** und ***ComponentDef*** definiert, wobei jeder Instanz von ***ComponentDependencyDef*** genau eine Instanz von ***ComponentDef*** zugeordnet wird und umgekehrt, eine Instanz von ***ComponentDef*** mehreren Instanzen von ***ComponentDependencyDef*** zugeordnet sein kann. Die zugehörige Semantik ist, daß jede Abhängigkeit exakt diejenige Softwarekomponente identifiziert, zu der die Abhängigkeit besteht und für jede Softwarekomponente natürlich mehrere andere Softwarekomponenten existieren können, die von dieser Softwarekomponente abhängig sind.

Die Strukturierung des Metamodells in zwei Assoziationen und eine zusätzliche Metaklasse zur Repräsentation der Abhängigkeitsrelation wurde hier gewählt, da für eine konkrete Abhängigkeit noch zusätzliche Informationen in einem Entwurfsmodell zu erfassen sind. Konkret erhält die Metaklasse ***ComponentDependencyDef*** ein Attribut des Namens ***local_dependency*** mit dem Typ ***boolean***. Ist in einem Entwurfsmodell für eine Instanz von ***ComponentDependencyDef*** der Wert dieses Attributes ***true***, so bedeutet dies, daß zur Ausführung des physikalischen Repräsentanten der durch die Assoziation ***dependencies*** mit der Instanz von ***CompoentDependencyDef*** verbundenen Instanz von ***ComponentDef*** der physikalische Repräsentant der mittels der Assoziation ***dependend_from*** verbundenen Instanz von ***ComponentDef*** auf der gleichen Maschine vorhanden sein muß. Ist der Wert des Attributes in einem Entwurfsmodell dagegen ***false***, so ist die Anwesenheit auf der gleichen Maschine nicht explizit gefordert und somit auch nicht Vorraussetzung für die Ausführbarkeit der Softwarekomponente.

Das Metamodell für ***ComponentDef*** mit dem Attribut ***properties*** und ***ComponentDependencyDef*** ist in Abb. 27 dargestellt.

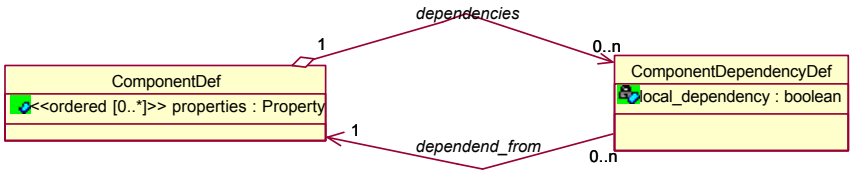


Abb. 27 Abhängigkeiten und Eigenschaften von Softwarekomponenten im Metamodell

3.5.2 Diskussion

Eine Werkzeugkette, die die Informationen über Softwarekomponenten auswertet und den *Deployment*-Vorgang auf der Grundlage dieser Information automatisiert ausführt, wird in einem EURESCOM Projekt unter Beteiligung des Autors erarbeitet [EU P924] - das hier entwickelte Metamodell dient diesem Projekt als konzeptionelle Basis.

Das beschriebene Metamodell kann leicht erweitert werden, um die in dem genannten Projekt oder anderweitig erarbeiteten eventuellen zusätzlichen Eigenschaften von Softwarekomponenten und ihren Abhängigkeiten zu berücksichtigen. Soll z.B. zusätzlich zu der bereits definierten maschinenbezogenen Lokalitätseigenschaft von Abhängigkeiten noch beschreibbar sein, daß eine Softwarekomponente die Anwesenheit einer anderen Softwarekomponente in dem gleichen Prozeß auf der ausführenden Maschine erfordert, so kann dies durch die Einführung eines zusätzlichen Attributes an der Metaklasse ***ComponentDependencyDef*** im Metamodell erfolgen.

Mit den bisher eingeführten Konzepten der *Deployment*-Sicht ist sowohl die Grundlage für die automatische Ableitung von Softwarekomponenten aus Entwurfsmodellen als auch - durch die Spezifikation von Eigenschaften und Abhängigkeiten von Softwarekomponenten - die Basis für einen werkzeugunterstützten *Deployment*-Vorgang geschaffen. Wird der *Deployment*-Vorgang auf der Grundlage der im Entwurfsmodell verfügbaren Informationen durchgeführt, so kann davon ausgegangen werden, daß auf den zur Ausführung der Softwarekomponenten vorgesehenen Maschinen die physikalischen Repräsentanten der Softwarekomponenten bereitgestellt und installiert sind. Die physikalischen Repräsentanten werden durch die Ableitungsregeln für die automatische Ableitung von Softwarekomponenten und einen anschließenden Implementierungsschritt (Bereitstellung des systemspezifischen Verhaltens - *Business Logic*) erzeugt. Ihre Ausführbarkeit auf der Maschine, auf der sie bereitgestellt und installiert wurden, ist gegeben, wenn die Eigenschaften und Abhängigkeiten von Softwarekomponenten im Entwurfsmodell vollständig und korrekt erfaßt sind und auch bei der Durchführung des *Deployment*-Vorganges berücksichtigt wurden. Zur Erfüllung des Systemzwecks müssen die Softwarekomponenten ausgeführt werden, die dann instanziierten COs erbringen durch ihr Verhalten und ihre Interaktionen den Systemzweck.

In verteilten Softwaresystemen ist es oftmals schwierig, die initiale Konfiguration, d.h. die Menge der initial vorhandenen Instanzen (COs) und ihrer initialen Bindungen aufzubauen (*Bootstrapping*). Eine Bindung zwischen COs ist dann aufgebaut, wenn Interfacereferenzen von unterstützten Interfacetypen einer Teilmenge von an der Bindung beteiligten COs den anderen an der Bindung beteiligten COs bekanntgemacht wurde. Die Schwierigkeit dieses Prozesses liegt u.a. daran, daß diese initialen Instanzen durch die Ausführung unterschiedlicher Softwarekomponenten entstehen und sich somit auch auf potentiell unterschiedlichen Maschinen befinden. Ein CO, das aufgrund seines eigenen Verhaltens versucht, eine Bindung zu einem anderen CO aufzubauen, kann nicht *a-priori* davon ausgehen, das dieses andere CO schon existiert.

Um das Problem der initialen Konfiguration eines verteilten Softwaresystems zu lösen, wird hier eine Beschreibung der Informationen über initiale Konfigurationen im Entwurfsmodell und eine werkzeugunterstützte Verarbeitung dieser Information mit dem Ziel des Aufbaus der initialen Konfiguration vorgeschlagen. Die Vorraussetzungen dafür sind durch die bisherigen Modellierungsschritte im Metamodell geschaffen. Notwendig ist insbesondere die Unterstützung der Modellierung der Konfigurationsmöglichkeiten für CO-Typen in der Konfigurationssicht.

3.5.3 Assemblage

In einem Entwurfsmodell ist zunächst eine Abstraktion für das verteilte Softwaresystem selbst zu erfassen. Die Notwendigkeit der Erfassung von solchen Abstraktionen für Softwaresysteme ist darin begründet, daß ein Entwurfsmodell Informationen und Elemente beinhalten kann, die zu mehr als einem Softwaresystem gehören. Dies ist u.a. der Fall, wenn ein Unternehmen Softwaresysteme modularisiert anbietet. Dann ist in einem Entwurfsmodell die Information über alle verfügbaren Module enthalten, während ein konkretes

Softwaresystem nur auf einer Teilmenge dieser Module basiert und somit auch nur eine Teilmenge aller Modellelemente für dieses Softwaresystem relevant ist.

Die Abstraktion eines realen Softwaresystems wird hier als Assemblage bezeichnet und in *CoRE$_{CEPT}$* aufgenommen. Der Term Assemblage weist darauf hin, daß dieses Konzept aus anderen Modellelementen komponiert wird. Diese Komposition erfolgt auf der Basis des Konzeptes CO-Typ. Eine Assemblage enthält also alle CO-Typen, deren Instanzen zur Erfüllung des Systemzwecks des Softwaresystems, daß durch die Assemblage im Entwurfsmodell beschrieben wird, beteiligt sind. Weiterhin gehört zu einer Assemblage die Beschreibung der initialen Konfiguration des Softwaresystems, also die Menge aller initial zu erzeugenden Instanzen der CO-Typen und der aufzubauenden Bindungen.

Im Metamodell wird das Konzept Assemblage durch eine neue Metaklasse ***AssemblyDef*** erfaßt. Diese Metaklasse ist von der abstrakten Metaklasse ***Container*** und damit auch von der abstrakten Metaklasse ***contained*** abgeleitet. Instanzen von ***AssemblyDef*** in einem Entwurfsmodell sind somit identifizierbar. Es wird gefordert, daß der definierende Container für eine Assemblagedefinition nur eine Instanz der Metaklasse ***NamespaceDef*** sein darf.

(Constraint 33) Für alle Instanzen von ***AssemblyDef*** in einem Entwurfsmodell gilt: Falls die Assoziation ***contains*** definiert ist, dann ist die korrespondierende Instanz der von ***Container*** abgeleiteten Metaklasse vom Typ ***NamspaceDef***.

Die Relation zwischen einer Assemblagedefinition und denjenigen CO-Typen, die für die Erfüllung des Systemzwecks des durch die Assemblagedefinition repräsentierten Softwaresystems benötigt werden, ist im Metamodell durch eine Assoziation ***contained_co_types*** zwischen den Metaklassen ***AssemblyDef*** und ***COTypeDef*** repräsentiert. Eine Instanz von ***AssemblyDef*** darf zu mehreren Instanzen von ***COTypeDef*** in Relation stehen, wobei umgekehrt eine Instanz von ***COTypeDef*** zu mehr als einer Instanz von ***AssemblyDef*** in Relation stehen darf. Das widerspiegelt die Tatsache, daß CO-Typen für die Erbringung von Verhalten im Kontext verschiedener Softwaresysteme wiederverwendet werden können. Zusätzlich ist die Navigation von einer Instanz der Metaklasse ***AssemblyDef*** zu denen durch die Assoziation ***contained_co_types*** zugeordneten Instanzen von ***COTypeDef*** durch das Metamodell ermöglicht.

Das Metamodell für Assemblagedefinitionen und den von diesen benötigten CO-Typen ist in Abb.28 dargestellt

.

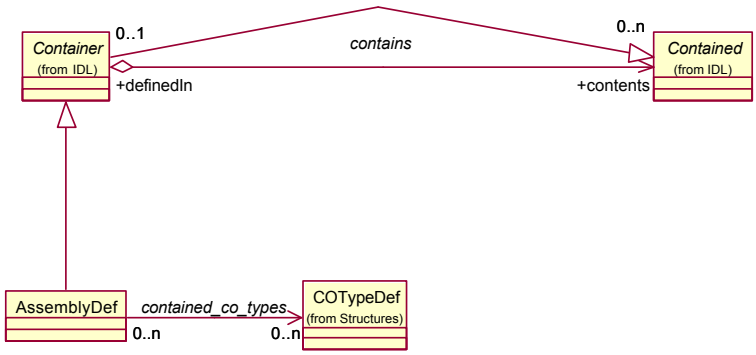


Abb. 28 Metamodell für Assemblage

3.5.4 Initiale COs

Entsprechend der aufgestellten Forderung nach einer Unterstützung für die initiale Konfiguration, ist ein Mechanismus für die Beschreibung der initialen COs ebenfalls in das Metamodell zu integrieren. Ein initiales CO ist eine Instanz eines CO-Typs, die zu Beginn der Ausführungszeit des Softwaresystems erzeugt wird. Dieser Mechanismus soll zudem die Modellierung einer großen Anzahl von COs in einer einfachen Weise erlauben. Es bieten sich zwei Verfahren an:

- die Definition von einzelnen und eindeutig identifizierbaren COs, oder
- die Definition von identifizierbaren Instanzmengen auf der Basis eines CO-Typs und der Information über die Anzahl der Instanzen in dieser Menge.

Die zweite Variante besitzt offensichtlich die Ausdruckskraft der ersten Variante (Anzahl der Instanzen in einer Menge ist eins). Die zweite Variante kann überall verwendet werden, wo die Modellierung einzelner, identifizierbarer Instanzen ein und desselben CO-Typs zeitaufwendig ist und zudem für alle diese Instanzen auch noch die gleiche Konfiguration vorgesehen ist, also die gleiche Menge von initialen Bindungen aufgebaut werden soll.

Dementsprechend wird für die Realisierung im Rahmen dieser Arbeit die zweite Variante ausgewählt.

Im Metamodell wird das Konzept initiale COs durch eine zusätzliche Metaklasse ***NamedCOSetDef*** repräsentiert. Diese Metaklasse ist von der abstrakten Metaklasse ***Contained*** abgeleitet, wodurch die Möglichkeit der Identifikation der Instanzen von ***NamedCOSetDef*** in einem Entwurfsmodell sichergestellt ist.

Als Container für Instanzen von ***NamedCOSetDef*** in einem Entwurfsmodell sind ausschließlich Instanzen der Metaklasse ***AssemblyDef*** erlaubt.

(Constraint 34) Für alle Instanzen von ***NamedCOSetDef*** in einem Entwurfsmodell gilt, daß die Assoziation ***contains*** definiert ist und zu einer Instanz von ***AssemblyDef*** führt.

Eine Instanzmenge von COs basiert immer auf einem einzigen CO-Typ. Dieser Sachverhalt ist im Metamodell durch die Definition einer Assoziation des Namens ***instantiated_from*** zwischen den Metaklassen ***COTypeDef*** und ***NamedCOSetDef*** erfaßt. Es ist außerdem festgelegt, daß in einem Entwurfsmodell eine Instanz von ***NamedCOSetDef*** immer zu genau einer Instanz von ***COTypeDefSet*** korrespondiert, umgekehrt einer Instanz von ***COTypeDef*** aber eine Menge von Instanzen von ***NamedCOSetDef*** zugeordnet sein kann. Zusätzlich ist die Navigation von einer Instanz von ***NamedCOSetDef*** zu der Instanz von ***COTypeDef*** möglich, die dieser durch die Assoziation ***instantiated_from*** zugeordnet ist.

Für die Festlegung der Anzahl von Instanzen des CO-Typs einer ***NamedCOSetDef***-Definition wird ein zusätzliches Attribut der Metaklasse ***NamedCOSetDef*** eingeführt. Der Name des Attributes ist ***initial_instances***, sein Typ ***short***. Das Metamodell für ***NamedCOSetDef*** ist in Abb.29 dargestellt.

Natürlich dürfen für ein Softwaresystem, das durch eine Instanz von ***AssemblyDef*** im Entwurfsmodell erfaßt ist, nur Instanzmengen von CO-Typen definiert werden, die Bestandteil des Softwaresystems sind.

(Constraint 35) Für alle Instanzen von ***AssemblyDef*** in einem Entwurfsmodell und alle Instanzen von ***NamedCoDefSet***, die über die Assoziation ***contains*** der Instanz von ***AssemblyDef*** zugeordnet sind, gilt: Die Assoziation ***insantiated_from*** führt zu einer Instanz von ***COTypeDef***, für die die Assoziation ***contained_co_types*** definiert ist und von der Instanz von ***AssemblyDef*** ausgeht.

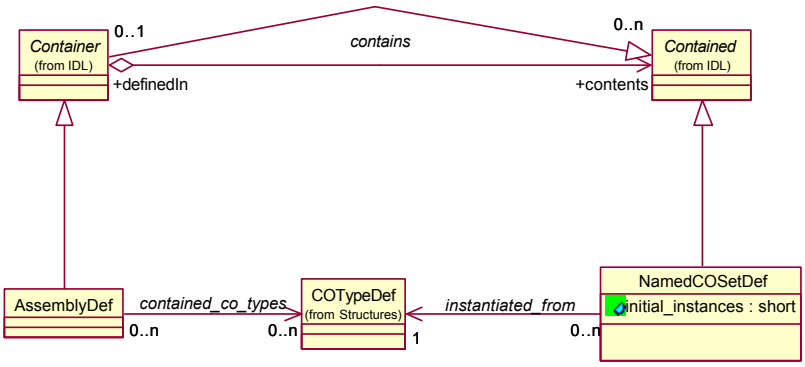


Abb. 29 Metamodell für Initiale COs

3.5.5 Initiale Bindung

Zur vollständigen Realisierung der Modellierung der initialen Konfiguration eines verteilten Softwaresystems muß nunmehr noch eine Möglichkeit zur Beschreibung der Bindungen zwischen COs bereitgestellt werden. Dazu dient das Konzept initiale Bindung.

Die Definition einer initialen Bindung umfaßt dabei die folgenden Informationen:

- die Menge von COs, die verbunden werden sollen,
- den Interfacetyp, der die Interaktionselemente für die Bindung definiert,
- die *Port*-Definitionen, die für die Herstellung der Bindung genutzt werden sollen.

An einer Bindung können mehrere Instanzen (COs) beteiligt sein. Dies ist insbesondere eine gültige Vorstellung, wenn der der Bindung zugrundeliegende Interfacetyp Interaktionselemente der nicht-operationalen Interaktionsarten enthält.

> *(Beispiel 3)* Im Falle von Signalinteraktion kann ein CO Sender von Signalen für eine Vielzahl von empfangenden COs sein.

Es ist zu beachten, daß die Beschreibung einer initialen Bindung in einem Entwurfsmodell noch nicht das Zustandekommen von tatsächlichen Interaktionen zwischen den an der Bindung beteiligten COs zur Ausführungszeit zur Folge haben muß. Statt dessen bedeutet diese Definition den Austausch von Interfacereferenzen über die angegebenen *Port*-Definitionen der beteiligten COs zum Zeitpunkt des Aufbaus der initialen Konfiguration.

Eine initiale Bindung basiert auf dem bereits eingeführten Konzept der initialen Instanzmengen von CO-Typen. Von jeder dieser Instanzmengen können ein oder mehrere *Port*-Definitionen des zu der Instanzmenge gehörigen CO-Typs in die Definition einer initialen Bindung einfließen. Dann werden zum Zeitpunkt des Aufbaus der initialen Bindung auf der Grundlage der Beschreibung im Entwurfsmodell Interfacereferenzen ausgetauscht. Dies erfolgt so, daß jede Instanz einer Instanzmenge mit einer *Provided-Port*-Definition, die Bestandteil der Definition einer initialen Bindung ist, mit jeder Instanz einer Instanzmenge mit einer *Used-Port*-Definition, die Bestandteil der Bindung ist, verbunden wird.

> *(Beispiel 4)* Die beschriebene Semantik ist in Abb.30 exemplarisch dargestellt. Bestandteil der Definition einer initialen Bindung seien drei Instanzmengen mit jeweils zwei initialen Instanzen der CO-Typen ***A***, ***B*** und ***C***. Für die Instanzmenge, die auf ***A*** basiert, sei die *Provided-Port*-Definition ***APP*** Bestandteil der Spezifikation der Bindung, für die Instanzmenge, die auf ***B*** basiert, die *Used-Port*-Definition ***BUP*** und für die Instanzmenge, die auf ***C*** basiert, die *Used-Port*-Definition ***CUP***. Dann werden alle Instanzen der Instanzmenge, die auf ***A*** basiert, mit allen Instanzen der Instanzmenge, die auf ***B***

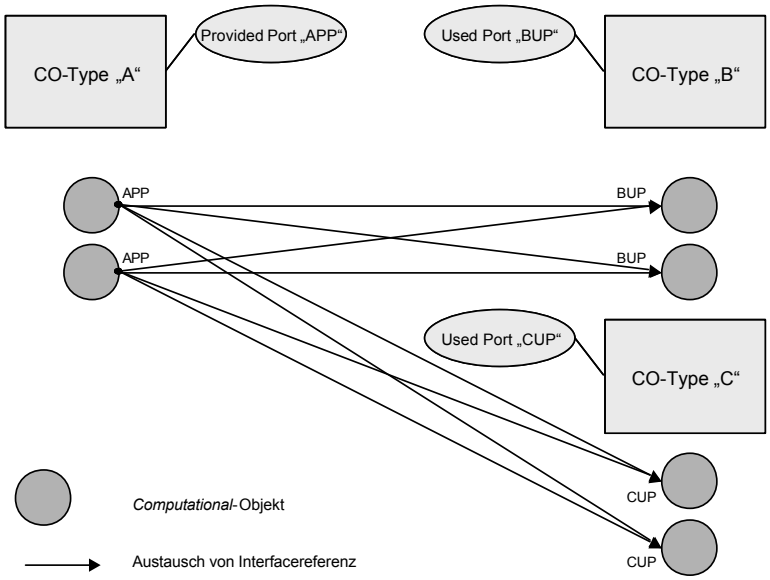


Abb. 30 Semantik von Initiale Bindung

basiert, und mit allen Instanzen der Instanzmenge, die auf ***C*** basiert, unter Benutzung der jeweiligen *Port*-Definitionen verbunden.

Im Metamodell wird für das Konzept initiale Bindung eine Metaklasse ***ConnectionDef*** eingeführt. Diese Metaklasse ist von der abstrakten Metaklasse ***Contained*** abgeleitet. Damit ist die eindeutige Möglichkeit der Identifikation von Instanzen von ***ConnectionDef*** in einem Entwurfsmodell gewährleistet. Als Spezialisierung der abstrakten Metaklasse ***Container***, deren Instanzen im Entwurfsmodell Instanzen der Metaklasse ***ConnectionDef*** enthalten dürfen, ist ausschließlich die Metaklasse ***AssemblyDef*** erlaubt.

(Constraint 36) Für alle Instanzen von ***ConnectionDef*** in einem Entwurfsmodell gilt, daß die Assoziation ***contains*** definiert sein muß und zu einer Instanz der Metaklasse ***AssemblyDef*** führt.

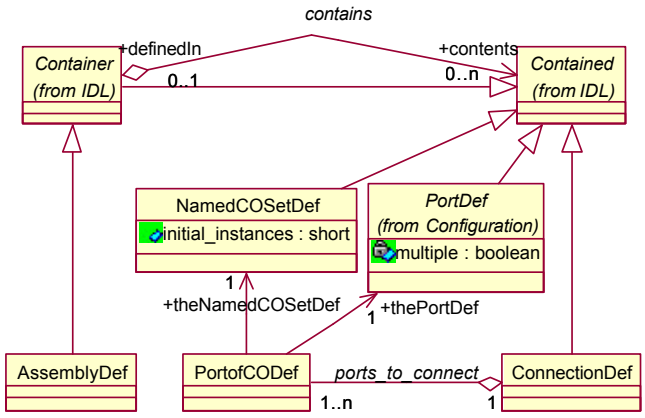


Abb. 31 Metamodell für Bindung

Es wird festgelegt, daß die Semantik von ***AssemblyDef*** als ***Container*** auf ***NamedCOSetDef***-Definitionen und Bindungsdefinitionen beschränkt ist.

(Constraint 37) Für alle Instanzen von ***AssmblyDef*** gilt: Falls eine Assoziation ***contains*** definiert ist, muß diese Assoziation zu einer Instanz von ***NamedCODefSet*** oder zu einer Instanz von ***ConnectionDef*** führen, wobei die Instanz von ***AssemblyDef*** die Rolle ***Container*** einnimmt.

Zur Modellierung derjenigen COs, die unter Benutzung von *Port*-Definitionen initial verbunden werden sollen, wird in das Metamodell die Metaklasse ***PortofCODef*** eingeführt. Diese Metaklasse besitzt Assoziationen zu den Metaklasse ***NamedCOSetDef*** und ***PortDef***, wobei jeder Instanz von ***PortofCODef*** genau eine Instanz der Metaklassen ***NamedCOSetDef*** und genau eine Instanz der Metaklasse ***PortDef*** zugeordnet werden.

Eine Instanz von ***PortofCODef*** in einem Entwurfsmodell bedeutet anschaulich, daß die durch diese Instanz bezeichnete Menge von COs verbunden werden soll, wobei zur Beschaffung bzw. Hinterlegung der Interfacereferenzen die durch diese Instanz bezeichnete *Port*-Definition des entsprechenden CO-Typs genutzt wird.

Im Metamodell ist die Metaklasse ***ConnectionDef*** mit der Metaklasse ***PortOfCODef*** durch eine Aggregation verbunden. Dabei ist festgelegt, daß jede Instanz von ***PortofCODef*** zu genau einer Instanz von ***ConnectionDef*** gehört und jede Instanz von ***ConnectionDef*** mehrere Instanzen von ***PortofCODef*** aggregieren kann. Potentiell können somit mehrere COs - u.U. verschiedenen Typs - verbunden werden.

Eine Bindungsdefinition enthält mindestens zwei Instanzen von ***PortofCODef***, wobei mindestens eine Instanz von ***PortOfCODef*** auf eine *Used-Port*-Definition verweist und eine andere auf eine *Provided-Port*-Definition. Der Grund hierfür besteht darin, daß sich Interfacereferenzen nur zwischen komplementären *Port*-Definitionen austauschen lassen.

(Constraint 38) Für alle Instanzen von ***ConnectionDef*** in einem Entwurfsmodell enthält die Menge der durch die Assoziation ***ports_to_connect*** aggregierten Instanzen von ***PortofCODef*** mindestens zwei Elemente.

(Constraint 39) Für alle Instanzen von ***ConnectionDef*** in einem Entwurfsmodell enthält die Menge der durch die Assoziation ***port_to_connect*** aggregierten Instanzen von ***PortofCODef*** mindestens eine Instanz für die die zugeordnete Instanz von ***PortDef*** vom Typ ***UsePortDef*** ist und mindestens eine Instanz für die die zugeordnete Instanz von ***PortDef*** vom Typ ***ProvidePortDef*** ist.

Weiterhin ist der Austausch von Interfacereferenzen nur möglich, falls die beteiligten *Port*-Definitionen auf denselben Interfacetyp verweisen und falls im Kontext der Instanzmenge die bezeichnete *Port*-Definition überhaupt zulässig ist, d.h. im Kontext des der Instanzmenge zugeordneten CO-Typs erfolgte.

(Constraint 40) Für alle Instanzen von ***ConnectionDef*** in einem Entwurfsmodell gilt, daß für alle von dieser Instanz aggregierten Instanzen von ***PortOfCODef*** die jeweils zugeordnete Instanz von ***PortDef*** auf ein und dieselbe Instanz von ***InterfaceDef*** verweisen muß.

(Constraint 41) Für alle Instanzen von ***PortOfCODef*** in einem Entwurfsmodell gilt, daß die zugeordnete Instanz von ***PortDef*** durch die Assoziation ***contains*** mit einer Instanz von ***COTypeDef*** verbunden ist, mit der auch die Instanz von ***NamedCOSetDef*** über die Assoziation ***instantiated_from*** verbunden ist, die ebenfalls der Instanz von ***PortOfCODef*** zugeordnet ist.

Die Konzepte der *Deployment*-Sicht sind auf der Grundlage von Konzepten der Struktur- und Konfigurationssicht definiert worden. Aus diesem Grund ergibt sich die Abhängigkeit zwischen den ***Package***-Definitionen ***Deployment*** und ***Structures*** sowie ***Configuration***.

3.5.6 Diskussion

Die Konzepte der *Deployment*-Sicht sind grundlegend für *CORE*. Nur durch die Information über die Softwarekomponenten und den von ihnen realisierten CO-Typen kann die beabsichtigte automatische Ableitung der Softwarekomponenten aus einem Entwurfsmodell erfolgen. Andererseits eröffnet die *Deployment*-Sicht durch das Konzept Assemblage die Möglichkeit, in einem Entwurfsmodell Informationen zu hinterlegen, die eine Automatisierung des *Deployment*-Vorgangs und den Aufbau der initialen Konfiguration des Softwaresystems durch Werkzeuge unter Verwendung einer geeigneten *Component-Support*-Plattform gestatten. Dieser Vorgang ist auch dann automatisiert durchführbar, wenn die Softwarekomponenten separat (z.B. von unterschiedlichen Herstellern) entwickelt wurden und nur ihre physikalischen Repräsentanten sowie die Modellinformation über das Softwaresystem (Assemblage) bekannt ist. Ist das Entwurfsmodell durch die Anwendung von *CORE* entstanden, so sichert die Verwendung der MOF-Technologie, daß die Entwurfsmodellinformation auf geeignete Weise, nämlich als XML-Dokument bereitgestellt werden kann. Das dazu notwendige Format (XML-DTD) ergibt sich automatisch durch Anwendung der im MOF-Standard beschriebenen Generierungsregeln für XML-DTDs aus MOF-konformen Metamodellen.

Ausgehend von diesen Betrachtungen kann nun eine Präzisierung der in [CoRE I], Kapitel1 gegebenen initialen Erläuterungen zur Entwicklung von Softwarekomponenten einschließlich des *Deployment*-Vorgangs erfolgen. In Abb.32 ist die Entwicklung unter Verwendung von *CORE* schematisch dargestellt.

Jedes Entwurfsmodell ist eine Instanz des MOF-konformen Metamodells. Ausgehend von diesem Entwurfsmodell können nun Softwarekomponenten automatisch erzeugt werden, wobei vor der Erzeugung bekannt sein muß, welche Technologien (*Component-Support*-Plattform, Programmiersprache,...) eingesetzt werden sollen. Nach der Fertigstellung der Softwarekomponenten durch Hinzufügen der systemspezifischen Funktionalität (*Business Logic*) werden physische Repräsentanten der Softwarekomponenten erzeugt und zu Paketen zusammengestellt (Schritt 1 in Abb.32). Zusätzlich wird auch das Systemmodell physikalisch in Form einer XML-Datei repräsentiert.

Die Modellierung und die automatische Ableitung von Softwarekomponenten sind natürlich iterative Schritte. Ein Entwurfsmodell unterliegt Veränderungen, bevor es den beabsichtigten Systemzweck optimal reflektiert. Die Softwarekomponenten, die durch die Anwendung von *CORE* entstehen, können auch separat entwickelt werden - in diesem Fall existiert für jede Softwarekomponente ein eigenes Teilmodell, das Entwurfsmodell des gesamten Softwaresystems wird dann aus diesen Teilen unter Hinzufügung zusätzlicher Informationen zusammengesetzt. Diese zusätzlichen Informationen sind gerade Definitionen von Assemblagen im Entwurfsmodell.

Nachdem alle Softwarekomponenten in Form physischer Repräsentanten verfügbar sind, kann der *Deployment*-Vorgang auf der Grundlage der ebenfalls verfügbaren Entwurfsmodellinformationen automatisch ausgeführt werden. Dazu werden zunächst alle benötigten Softwarekomponenten auf für die Ausführung geeigneten Maschinen bereitgestellt (Schritt 2 in Abb.32). Auf der Basis des Entwurfsmodells können die Maschinen unter Beachtung der Eigenschaften der Softwarekomponenten automatisch ausgewählt werden. Zu diesem Zweck wird vorausgesetzt, daß Informationen über die Eigenschaften der Maschinen ebenfalls verfügbar sind, die durch eine geeignete *Component-Support*-Plattform bereitgestellt werden.

Nach der Bereitstellung der Softwarekomponenten kann (wiederum auf der Basis von Entwurfsmodellinformationen über die Softwarekomponenten) deren Installation erfolgen (Schritt 3 in Abb. 32). Anschließend können die Softwarekomponenten auf den Maschinen ausgeführt werden - die Grundlage für den Aufbau der initialen Konfiguration ist geschaffen.

Die initiale Konfiguration wird entsprechend der Auswertung von Entwurfsmodellinformationen (Assemblagedefinitionen) durch das Erzeugen von COs und Bindungen hergestellt (Schritt 4 in Abb. 32).

Bezüglich der in [CoRE I], Kapitel1 aufgestellten Anforderungen an Entwicklungstechniken läßt sich feststellen, daß mit der Definition der Konzepte der *Deployment*-Sicht in Kombination mit den Konzepten der anderen Sichten ein Schritt zur Realisierung der Anforderungen nach flexibler Integration von Softwaresystemkomponenten und kurze Entwicklungszeiten (vgl. [CoRE I], Abschnitt 1.5) getan ist. Komponenten lassen sich nun auf der Grundlage ihrer Entwurfsmodellinformationen zu ausführbaren Softwaresystemen zusammensetzen, die Konfiguration der in den einzelnen Komponenten realisierten COs läßt sich im Entwurfsmodell beschreiben und unter ausschließlicher Verwendung dieser Information automatisch aufbauen. Durch den modularen Aufbau von Softwaresystemen aus (wiederverwendbaren) Softwarekomponenten ist die Voraussetzung für kurze Entwicklungszeiten durch einen hohen Wiederverwendungsgrad gegeben.

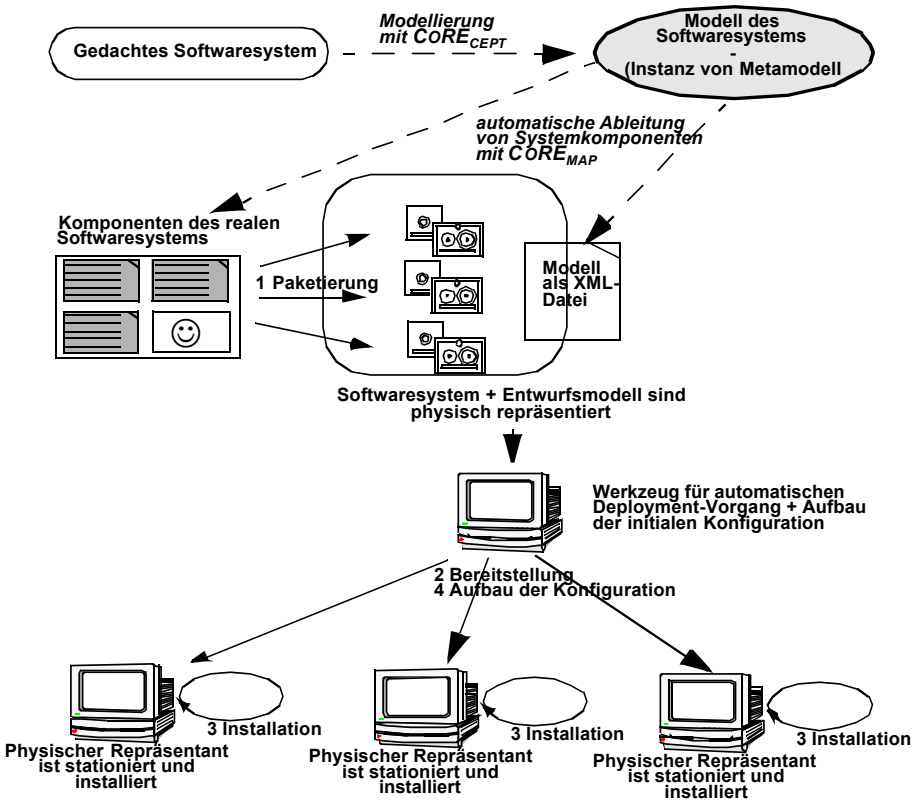


Abb. 32 Von Systemmodellen zu Softwaresystemen, die ausgeführt werden

3.6 Interaktionssicht

Die Konzepte der bisher eingeführten Sichten berücksichtigen die in [CoRE I], Kapitel 1 aufgestellten Anforderungen bezüglich der Modellierung von verteilten Softwaresystemen. Eine Ausnahme bildet bisher jedoch die für die Telekommunikationsdomäne wichtige Anforderung nach Unterstützung von Gütebeschreibung und Garantie.

Bezogen auf die Modellierung bedeutet dies, daß geeignete Konzepte und Relationen für die Erfassung von Güteeigenschaften und auf diesen Eigenschaften basierenden Güteanforderungen in *CORE$_{CEPT}$* zu integrieren sind, um auf der Grundlage der dann möglichen Beschreibungen in einem Entwurfsmodell eine Unterstützung durch eine geeignete *Component-Support*-Plattform zu ereichen.

Die Anforderung nach Gütebeschreibung und Garantie wird in *CORE* dadurch gewährleistet, daß die Gütebeschreibungen durch die Modellierung unterstützt werden wohingegen die Gütegarantie durch eine geeig-

nete *Component-Support*-Plattform und entsprechende Ableitungsregeln für die automatische Ableitung von für diese Plattform geeigneten Softwarekomponenten erfolgt.

Der hier gewählte MOF-basierte Ansatz zur Konstruktion von $CORE_{CEPT}$ ist offen für derartige Erweiterungen, da ein objektorientiertes Verfahren zur Modellierung der Konzepte und Relationen eingesetzt wurde. Konzepte und Relationen für die Unterstützung von Gütebeschreibungen lassen sich durch Verwendung und Spezialisierung bereits vorhandener Konzepte in $CORE_{CEPT}$ integrieren.

Es ist nicht das Ziel dieser Arbeit, neue Konzepte für die Beschreibung von Güteeigenschaften und -anforderungen zu entwickeln, da einerseits verschiedene Ansätze existieren und andererseits die Komplexität des Themas eine eigenständige Arbeit rechtfertigen [vHalt 00]. Güteeigenschaften und -anforderungen werden in der Literatur als *Quality-of-Service*-Eigenschaften und -anforderungen (QoS) bezeichnet. *Quality-of-Service*-Anforderungen können dabei als sehr weitreichende Menge von Anforderungen verstanden werden. *Performance*- und Skalierbarkeitsanforderungen gehören ebenso dazu wie Sicherheits- oder Verfügbarkeitsaspekte. Es wird gezeigt, auf welche Weise einer der existierenden Ansätze für die Beschreibung von QoS-Anforderungen im Metamodell konzeptionell erfaßt und mit den bisherigen Konzepten der bereits beschriebenen Sichten integriert werden kann.

3.6.1 QML zur Modellierung von QoS-Anforderungen

Einer der bereits existierenden Ansätze zur Modellierung von QoS-Aspekten ist die Sprache *Quality Modeling Language* (QML, [FK 98]). QML wurde von der Firma *Hewlett Packard* entwickelt, um QoS-Anforderungen von COs in verteilten Softwaresystemen beschreiben zu können. Die Konzepte von QML werden hier exemplarisch für eine Integration in $CORE_{CEPT}$ ausgewählt, da sie einerseits analog zu den bereits in $CORE_{CEPT}$ integrierten Konzepten speziell für verteilte Softwaresysteme entwickelt sind, deren Softwarekomponenten unter Verwendung einer geeigneten *Component-Support*-Technologie erstellt wurden. Andererseits erfolgt die Beschreibung der QoS-Anforderungen in QML interfaceorientiert, womit eine kanonische Integration mit den bereits vorgestellten Konzepten und Relationen von $CORE_{CEPT}$ ermöglicht wird. Zudem ist die Ausdrucksfähigkeit von QML geeignet, um die Forderung nach Beschreibung von QoS-Eigenschaften und -anforderungen zu erfüllen, wie u.a. im Rahmen eines EURESCOM Projektes [EU P924] und in laufenden Forschungsprojekten bei *KPN Research* [vHalt 00] gezeigt wird.

QML stellt Konzepte und eine Notation bereit, um QoS-Anforderungen von Softwarekomponenten beschreiben zu können. Präzise formuliert werden Anforderungen von Softwarekomponenten durch Anforderungen an von den Softwarekomponenten realisierten Interfacetypen erfaßt. Durch die Bereitstellung der Konzepte und der Notation ist aber noch kein Mechanismus zur Realisierung der QoS-Anforderungen durch eine *Component-Support*-Plattform und entsprechende Ableitungsregeln für die automatische Erzeugung von Softwarekomponenten impliziert.

QML ist analog zu CORBA-IDL eine deklarative Sprache, sie kann als Ergänzung zu CORBA-IDL verstanden werden, wobei in CORBA-IDL-Spezifikationen die Signaturen von Interfaces beschrieben werden, während in QML-Spezifikationen QoS-Anforderungen für die Benutzung der an Interfaces bereitgestellten Operationen und Attribute erfaßt sind. Zur Integration der QML-Konzepte in $CORE_{CEPT}$ ist die Beschreibung der QoS-Anforderungen auf alle Interaktionsarten im Kontext des Konzeptes Interfacetyp auszudehnen.

Die Basiskonzepte von QML sind QoS-Kontrakttypen, QoS-Kontrakte und QoS-Profile.

- *Contract Type*

  Ein Kontrakttyp wird benutzt, um eine Menge von QoS-Eigenschaften einer bestimmten Kategorie zu definieren. Eine solche Kategorie ist z.B. *Performance.* Ein Kontrakttyp legt eine Menge von QoS-Eigenschaften fest, wobei jede Eigenschaft benutzt werden kann, um eine QoS-Anforderung zu definieren. Beispiel für eine QoS-Eigenschaft der Kategorie *Performance* ist die Antwortzeit für eine Operation - eine QoS-Anforderung für diese Eigenschaft kann dann die Forderung nach einer Anwortzeit unterhalb von

5 Sekunden sein. Jede in einem Kontrakttyp angegebene Eigenschaft wird in QML als Dimension bezeichnet.

- *Contract*

  Ein Kontrakt ist eine Instanz eines Kontrakttyps und repräsentiert eine QoS-Beschreibung einer bestimmten Kategorie. Praktisch bedeutet dies, daß für jede durch den QoS-Kontrakttyp spezifizierte QoS-Eigenschaft eine Anforderung bezüglich dieser Eigenschaft unter Angabe eines Wertes formuliert wird. Je nach Art der Eigenschaft wird gefordert, daß dieser Wert entweder überschritten, unterschritten oder genau erfüllt sein muß. Zusätzlich lassen sich in QML statistische Elemente für die Formulierung von Anforderungen bei der Definition von Kontrakten einsetzen. Beispiel für ein statistisches Element ist die Anforderung, daß ein bestimmter Wert einer QoS-Eigenschaft in 80% der Fälle erreicht werden muß. Kontrakte in QML sind Spezifikationen von QoS-Anforderungen, die zunächst noch unabhängig von Interfacedefinitionen sind. Sie können aber mit bestimmten Interfacedefinitionen verbunden werden. Dazu dient das Konzept Profile.

- *Profile*

  Ein Profil (*Profile*) beschreibt die Bindung eines Kontraktes (und damit die Beschreibung einer QoS-Anforderung basierend auf einem Kontrakttyp) mit einem bestimmten Interfacetyp. Dabei ist es erlaubt, einem Interfacetyp mehr als einen Kontrakt zuzuordnen, z.B. kann einem Interfacetyp gleichzeitig ein Kontrakt der Kategorie *Performance* und ein Kontrakt der Kategorie Verfügbarkeit zugeordnet werden. Eine *Profile*-Definition enthält außerdem die Zuordnung von Kontrakten zu spezifischen Interaktionselementen an einem Interfacetyp mit der Semantik, daß der Kontrakt dann nur für Interaktionen bezüglich des angegebenen Interaktionselements erfüllt werden muß.

In Abb.33 sind die QML-Sprachkonzepte und ihr Verwendungszweck schematisch dargestellt.

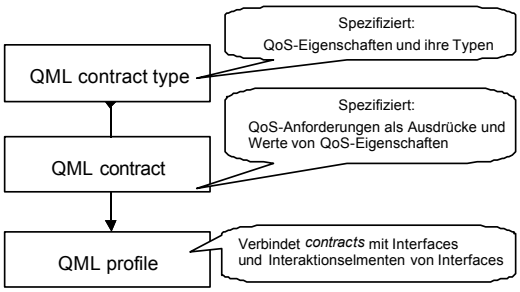


Abb. 33 QML-Sprachkonzepte

Im folgenden werden die Konzepte Kontrakttyp, Kontrakt und Profil in *CoRE$_{CEPT}$* integriert. Damit soll erreicht werden, daß die Beschreibung von QoS-Anforderungen auch für nicht-operationale Interaktionselemente möglich ist und daß die Zuordnung von QoS-Kontrakten zu Interfacedefinitionen flexibilisiert wird. Flexibilisierung bedeutet einerseits, daß QoS-Kontrakte nicht mehr an Interfacetypen, sondern an Bindungen, die auf diesen Interfacetypen basieren, gekoppelt werden. Andererseits wird Flexibilisierung durch die Anwendung eines fallbasierten Ansatzes erreicht, der bereits in [CoRE I], Kapitel3 im Überblick vorgestellt wurde.

3.6.2 Metamodell für QoS-Eigenschaften und -anforderungen

Konzeptionell sind QoS-Kontrakte als eine zur Ausführungszeit auszuhandelnde Menge von Güteeigenschaften für Bindungen zwischen COs zu verstehen, wobei ein oder mehrere COs die Serverrolle und ein oder mehrere COs die Klientenrolle einnehmen. Die Verhandlung solcher Kontrakte und ihre Gewährlei-

stung für die Interaktionen zwischen den beteiligten COs sind Aufgaben der *Component-Support*-Plattform. Um eine *Component-Support*-Plattform in die Lage zu versetzen, die geforderte Funktionalität zu erbringen, sind im Entwurfsmodell mindestens die folgenden Informationen zu hinterlegen:

- Welche sind die im Kontext des modellierten Softwaresystems relevanten Mengen von Güteeigenschaften (Typen von Kontrakten)?
  Ohne eine Information über die Art der Eigenschaften können natürlich auch keine weitergehenden Aussagen und Anforderungen formuliert werden. Typen von Kontrakten nehmen für die Formulierung von Güteeigenschaften und -anforderungen eine ähnlich grundlegende Rolle ein, wie Datentypen für die Formulierung von konkreten Daten.
- Was sind bezogen auf eine Güteeigenschaft bessere oder schlechtere Werte?
  Es muß zur Realisierung einer Verhandlung, bei der bestimmte Güteanforderungen zu berücksichtigen sind, festgelegt werden, welche Werte die Anforderungen erfüllen. Dabei ist es meist zulässig, die Anforderungen zu übertreffen, was die Verfügbarkeit von Vergleichsoperatoren für Güteeigenschaften bedingt.
- Was sind, bezogen auf eine potentielle Bindung, die minimalen Güte-Anforderungen der beteiligten COs, die die Klientenrolle einnehmen?
  Die Anforderungen von Klienten an eine Bindung können entweder schon zur Entwicklungszeit des Softwaresystems bekannt sein, oder aber während der Ausführungszeit des Softwaresystems entstehen. Im ersten Fall können die Anforderungen in einem Entwurfsmodell erfaßt werden. Diese sind dann die Basis für spezielle Ableitungsregeln für die automatische Erzeugung von Softwarekomponenten.

3.6.3 Kontrakttyp

Das Konzept Kontrakttyp wird verwendet, um in Entwurfsmodellen zu beschreiben, welche Arten von Güteeigenschaften für die Aushandlung und den Aufbau eines konkreten Kontraktes, der auf einem bestimmten Kontrakttyp basiert, zur Ausführungszeit berücksichtigt werden. Dieses wird durch die ebenfalls zu erfassenden Güteanforderungen für Bindungen im Entwurfsmodell festgelegt.
Das Konzept Kontrakttyp korrespondiert zu dem aus QML bekannten Prinzip *Contract Type*, und ist hier zunächst unabhängig von einem Interfacetyp bzw. CO-Typ, für dessen spätere Bindungen der Kontrakttyp verwendet werden soll. Ein Kontrakttyp enthält Kontraktelemente, die wiederum analog zu QML als Dimensionen bezeichnet werden. Jede Dimension ist im Kontext eines Kontrakttyps eindeutig identifizierbar und besitzt einen Typ, der entweder numerischer Art oder ein Aufzählungstyp ist. Beispiele für konkrete numerische Dimensionen sind Antwortzeiten von Operationen, Verzögerungen für die Übertragung von Signalen oder Bandbreiten von *Continous-Media*-Interaktionen. Beispiel für eine Dimension, die auf einem Aufzählungstyp basiert, ist der Sicherheitsgrad für operationale Kommunikation.
Dimensionen spezifizieren außerdem aus den bereits motivierten Gründen, ob Werte der Dimension in aufsteigender oder absteigender Reihenfolge bessere Güteeigenschaften bedeuten.

Im Metamodell ist das Konzept Kontrakttyp durch eine Metaklasse ***QoSContractTypeDef*** modelliert. Diese Metaklasse ist von der abstrakten Metaklasse ***Contained*** abgeleitet. Damit ist die eindeutige Möglichkeit der Identifikation von Kontrakttypdefinitionen in einem Entwurfsmodell gewährleistet. Es wird gefordert, daß Kontrakttypdefinitionen nur im Kontext einer Namensraumdefinition vorgenommen werden dürfen.

(Constraint 42) Für alle Instanzen von ***QoSContractTypeDef*** in einem Entwurfsmodell gilt: Falls die Assoziation ***contains*** definiert ist, führt sie zu einer Instanz der Metaklasse ***NamespaceDef***.

Ein Kontrakttyp enthält eine Menge von identifizierbaren Dimensionen. Diese Situation ist im Metamodell durch ein entsprechendes Attribut der Metaklasse ***QoSContractTypeDef*** erfaßt, der Name des Attributes ist ***members***, der Typ eine geordnete, nichtleere Liste von Elementen des Typs ***DimensionField***. ***DimensionField*** ist ebenfalls eine Metaklasse, die von der abstrakten Metaklasse ***Typed*** abgeleitet ist. Durch die Verwendung des

Modellierungsmusters *Type-Typed* an dieser Stelle ist es möglich, in Entwurfsmodellen den Typ einer Dimension in Form einer Instanz einer von der Metaklasse ***IDLType*** abgeleiteten Klasse festzulegen, es sind nur numerische und Aufzählungstypen zugelassen.

(Constraint 43) Für alle Instanzen von ***DimensionField*** in einem Entwurfsmodell gilt, daß die Assoziation ***IDLType-Typed*** zu einer Instanz von ***AliasDef***, ***PrimitiveDef*** oder ***EnumDef*** führen muß. Im Falle von ***PrimitiveDef*** gilt zusätzlich, daß der Wert des Attributes ***primitive_kind*** ***PK_SHORT***, ***PK_LONG***, ***PK_USHORT***, ***PK_ULONG***, ***PK_FLOAT***, ***PK_DOUBLE***, ***PK_LONGLONG***, ***PK_ULONGLONG*** oder ***PK_LONGDOUBLE*** sein muß. Im Falle von ***AliasDef*** ist diese Regel rekursiv auf die Instanz von ***AliasDef*** anzuwenden.

Die in einer Kontrakttypdefinition zusammengefaßten Dimensionen sollen identifizierbar sein, um späterhin die Formulierung von Anforderungen auf der Basis dieser Dimensionen zu gestatten. Zu diesem Zweck wird ein Attribut ***identifier*** des Typs ***string*** für die Metaklasse ***DimensionField*** eingeführt.

Es wird gefordert, daß alle in der Liste ***members*** enthaltenen Elemente *eindeutig* identifizierbar sind.

(Constraint 44) Für alle Instanzen von ***QoSContractTypeDef*** in einem Entwurfsmodell gilt, daß für alle in der Liste ***members*** enthaltenen Instanzen der Metaklasse ***DimensionField*** die Werte der Attribute ***identifier*** paarweise verschieden sein müssen.

Die Information darüber, ob aufsteigende oder absteigende Werte für Dimensionen bessere Werte im Sinne der beschriebenen Güteeigenschaft darstellen, ist in einem Entwurfsmodell an Instanzen der Metaklasse ***DimensionField*** durch die Belegung eines Attributes mit dem Namen ***direction*** zu spezifizieren. Dieses Attribut ist an der Metaklasse ***DimensionField*** definiert und hat als Typ die *Enumeration*-Definition ***Direction***. Diese besitzt die *Enumerator*-Elemente ***increasing*** und ***decreasing***. Das *Enumerator*-Element ***increasing*** bedeutet, daß größere Werte der der Instanz von ***DimensionField*** zugeordneten Instanz von ***IDLType*** bessere Güte darstellen. Das *Enumerator*-Element ***decreasing*** hingegen bedeutet, daß kleinere Werte der zugeordneten Instanz von ***IDLType*** bessere Güte darstellen.

Das vollständige Metamodell für QoS-Kontrakttypen ist in Abb. 34 dargestellt.

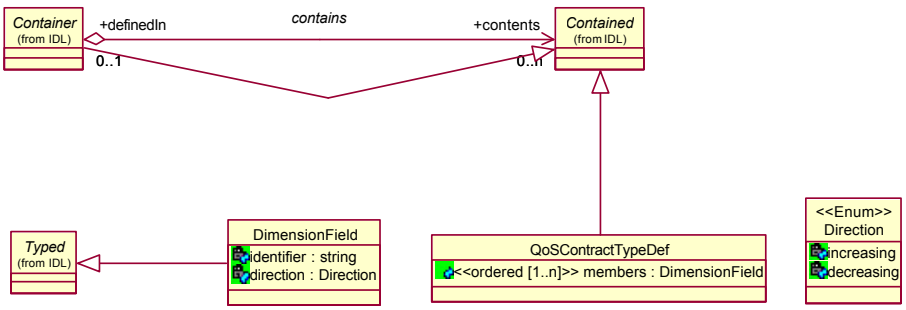


Abb. 34 Metamodell für QoS-Kontrakttyp

Wie bereits ausgeführt, dienen QoS-Kontrakttypen zur Formulierung von Anforderungen, die Klienten an eine potentielle Interaktion im Rahmen einer Bindung mit Servern haben können. Um diejenigen Kontrakttypen zu spezifizieren, die für die Formulierung von Anforderungen im Rahmen einer Bindung tatsächlich benutzt werden können, wird hier eine Relation zwischen Interfacetypen und QoS-Kontrakttypen eingeführt. Die Semantik dieser Relation ist, daß ein konkreter Kontrakttyp für die Formulierung von Anforderungen an eine Bindung benutzt werden kann, wenn zwischen dem Kontrakttyp und dem Interfacetyp, der der Bindung zugrundeliegt, eine solche Relation definiert wurde.

Im Metamodell ist zu diesem Zweck eine Aggregation zwischen den Metaklassen ***EnhancedInterfaceDef*** und ***QoSContractTypeDef*** eingeführt, die den Namen ***qos_contract_types*** erhält. Es wird festgelegt, daß einer Instanz von ***EnhancedInterfaceDef*** in einem Entwurfsmodell mehrere Instanzen von ***QoSContractTypeDef*** zugeordnet werden können und daß eine Instanz von ***QoSContractType*** zu mehreren Instanzen von ***EnhancedIterfaceDef*** zugeordnet sein kann. Von einer Instanz von ***EnhancedInterfaceDef*** kann ferner zu den zugeordneten Instanzen von ***QoSContractTypeDef*** navigiert werden. Diese Situation ist in Abb. 35 dargestellt.

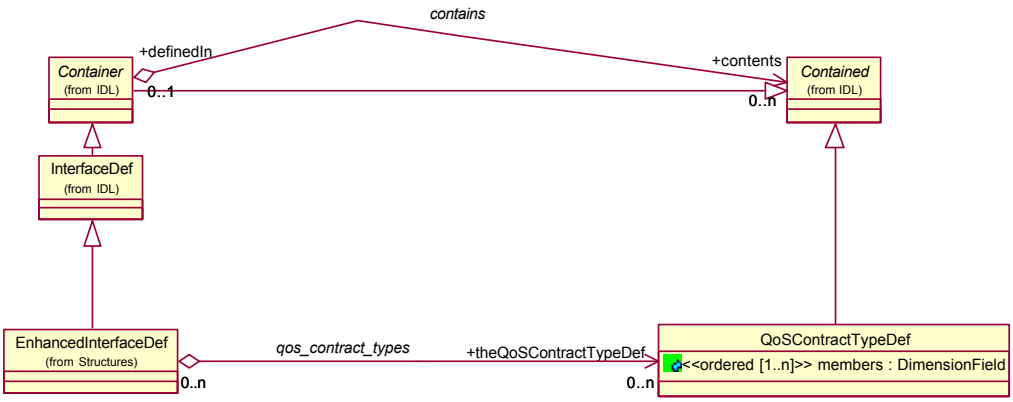


Abb. 35 Metamodell für Erweiterter Interfacetyp

3.6.4 Bindung mit Regel

Mit den oben eingeführten Konzepten und Relationen sind nunmehr die Vorraussetzungen geschaffen, um in einem Entwurfsmodell Anforderungen von Klienten an potentielle Interaktionen im Rahmen einer Bindung mit einem Server spezifizieren zu können. Diese Anforderungen werden unter Benutzung derjenigen Kontrakttypen spezifiziert, die dem Interfacetyp zugeordnet wurden, der der jeweiligen Bindung zugrundeliegt. Eine derartige Anforderung wird hier definiert als ein logischer Ausdruck über die in den Kontrakttypen spezifizierten Dimensionen. Zur Definition dieses Ausdrucks kann eine beliebige Notation benutzt werden, ein Kandidat für eine solche Notation ist *Object Constraint Language (*OCL) [OMG UML1.3], wie im folgenden Beispiel illustriert.

> *(Beispiel 5)* Sei in einem Entwurfsmodell der Kontrakttyp ***Performance*** spezifiziert, der zwei Dimensionen ***rate*** und ***delay*** mit dem zugehörigen Datentyp ***long*** enthält. Sei ferner ***Performance*** über die Aggregation ***qos_contract_types*** mit einem Interfacetyp ***I*** verbunden. Dann kann die folgende Anforderung eines Klienten an eine Bindung, die auf ***I*** basiert, unter Benutzung von OCL definiert werden:
> **{context Performance: delay < 5 and rate > 100[1]}**

Mit dieser Anforderung ist festgelegt, daß Klienten für Interaktionen unter Benutzung der Bindung eine Verzögerung erwarten, die kleiner als 5 ist und eine Aufrufrate, die über 100 liegt.

Um die Voraussetzung für eine Spezifikation von Anforderungen an eine Bindung zu schaffen, wird im Metamodell eine Spezialisierung der Metaklasse ***ConnectionDef*** (vgl. Abschnitt 3.5.5) vorgenommen. Die gegenüber dem Konzept initiale Bindung erweiterte Semantik besteht in der Möglichkeit, den Instanzen dieser Spezialisierung einen logischen Ausdruck zur Beschreibung der Anforderungen an eine Bindung im Entwurfsmodell hinzuzufügen.

1. Die Maßeinheit ist hier zusätzliche Metainformation.

Konkret wird im Metamodell die neue Metaklasse ***ConstrainedConnectionDef*** als Spezialisierung der Metaklasse ***ConnectionDef*** definiert. Diese Metaklasse bekommt ein Attribut des Namens ***client_req*** vom Typ ***string***. An Instanzen von ***ConstrainedConnectionDef*** kann durch die Belegung dieses Attributes ein logischer Ausdruck in einer geeigneten Notation zugordnet werden. Allerdings muß zur Auswertung dieses Ausdrucks noch die tatsächlich genutzte Notation im Entwurfsmodell spezifiziert werden. Dazu erhält die Metaklasse ***ConstrainedConnectionDef*** ein weiteres Attribut ***request_language*** vom Typ ***string***. Das Metamodell für ***ConstrainedConnectionDef*** ist in Abb. 36 dargestellt.

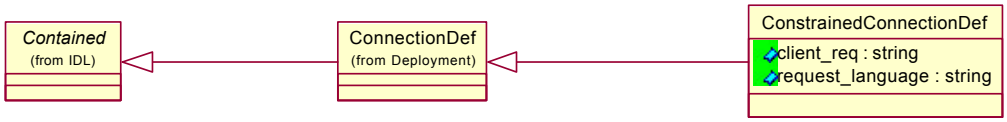


Abb. 36 Metamodell für Bindung mit Regel

3.6.5 Prädikat

In [CoRE I], Kapitel3 wurde bereits diskutiert, daß die Spezifikation von Klientenanforderungen an eine Bindung fallbasiert erfolgen muß. Die bisher im Metamodell vorgenommene Modellierung der Metaklasse ***ConnectionDef*** erlaubt allerdings noch keinen fallbasierten Ansatz, da die Menge der im Rahmen der Bindung zu benutzenden *Port*-Definitionen auf der Basis von Instanzmengen von COs angegeben wird (vgl. Konzept 3.5.4 und Konzept 3.2.9).

Zur Realisierung des fallbasierten Ansatzes ist somit eine zusätzliche Möglichkeit zur Definition von Instanzen von CO-Typen vorzusehen. Steht diese zur Verfügung, können die darauf aufbauenden Konzepte (Metaklassen ***PortOfCODef*** und ***ConnectionDef***) wiederverwendet werden.

Zur Modellierung der Definition von Instanzen von CO-Typen bietet sich die Verwendung der objektorientierten Paradigmen des MOF-Meta-Metamodells an. Zunächst kann die Definition von ***PortofCODef*** verändert werden. Dazu wird eine neue abstrakte Metaklasse ***COSetDef*** eingeführt, die von der abstrakten Metaklasse ***Contained*** abgeleitet ist. Die Metaklasse ***PortofCODef*** erhält anstelle der bisher verwendeten Assoziation zu ***NamedCODefSet*** eine Assoziation mit den gleichen Eigenschaften zu der neu eingeführten Metaklasse ***COSetDef***. Die Assoziation ***instantiated_from***, die bisher zwischen ***COTypeDef*** und ***NamedCOSetDef*** definiert ist, führt nun ebenfalls zur Metaklasse ***COSetDef***. Von der Metaklasse ***COSetDef*** können im Entwurfsmodell allerdings keine Instanzen spezifiziert werden, es ist also eine Spezialisierung vorzunehmen. Diese Spezialisierung wird zunächst für die Metaklasse ***NamedCOSetDef*** eingeführt. Jedes weitere Verfahren zur Definition von Instanzmengen von COs in einem Entwurfsmodell kann im Metamodell nun ebenfalls als Spezialisierung von ***COSetDef*** vorgenommen werden.

Die beschriebene Veränderung des Metamodells ist in Abb.37 dargestellt.

Es sei darauf hingewiesen, daß mit den hier eingeführten Veränderungen die im Rahmen der Diskussion der *Deployment*-Sicht eingeführte Semantik der Konzepte von initialen Instanzmengen und initialen Bindungen nicht verändert wird. Die zusätzliche Generalisierung der Metaklasse ***NamedCOSetDef*** durch ***COSetDef*** im Metamodell ist für die Modellierung und damit für die Definition von Entwurfsmodellen transparent. Die formulierten *Constraint*-Definitionen bleiben ebenfalls gültig, da alle Assoziationen und Spezialisierungsrelationen, die vorher für die Metaklasse ***NamedCOSetDef*** definiert waren, nun an der Metaklasse ***COSetDef*** definiert sind. Durch die Spezialisierungsrelation zwischen ***NamedCOSetDef*** und ***COSetDef*** sind sie aber weiterhin im Kontext von ***NamedCOSetDef*** gültig. *Constraint*-Definitionen, die sich auf Assoziationen und Spezialisierungen beziehen, gelten somit weiterhin uneingeschränkt.

Das Konzept der fallbasierten Definition von Instanzmengen von COs für Bindungen kann ebenfalls als Spezialisierung der Metaklasse ***COSetDef*** definiert werden. Wie bereits in [CoRE I], Kapitel3 eingeführt, beruht dieses Verfahren auf Prädikaten, also auf der Spezifikation von zu wahr oder falsch evaluierbaren

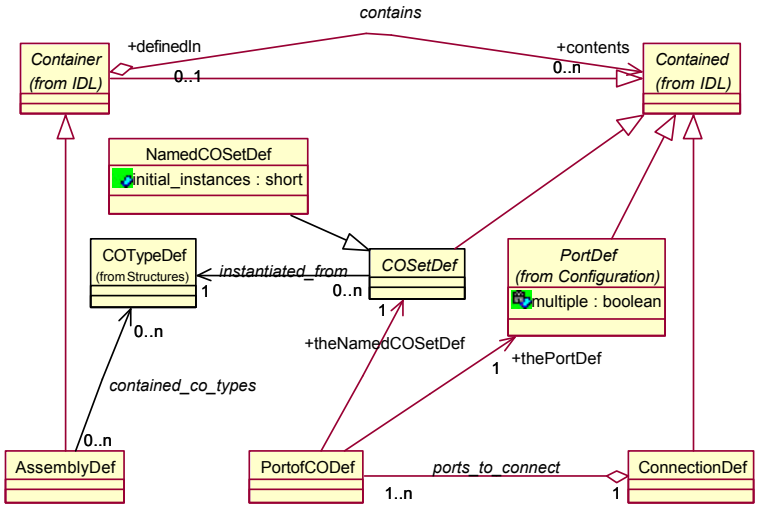


Abb. 37 Verändertes Metamodell für Instanzmengen von COs

Aussagen im Kontext von CO-Typen. Zur Definition der prädikatbasierten Instanzmengen von CO-Typen ist zuerst eine geeignete Definition der Prädikate selbst vorzunehmen.

Ein Prädikat ist eine im Kontext eines CO-Typen definierte, namentlich identifizierbare Aussage, die im Kontext eines COs zur Ausführungszeit zu wahr oder falsch evaluierbar ist. Im Metamodell werden Prädikate als neue Metaklasse ***PredicateDef*** eingeführt, die als Spezialisierung der abstrakten Metaklasse ***Contained*** eingeführt wird. Damit ist die eindeutige Möglichkeit der Identifikation von Prädikatdefinitionen im Kontext des definierenden Containers sichergestellt. Das Metamodell für Prädikate ist in Abb. 38 dargestellt.

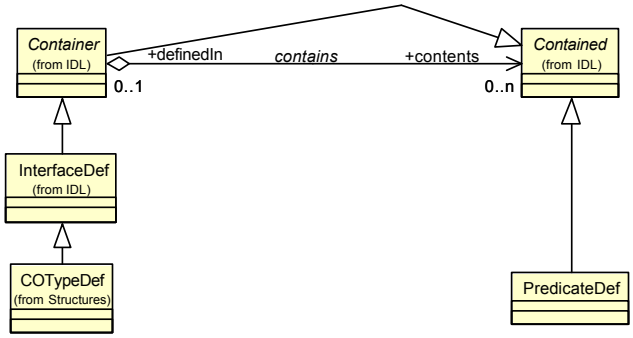


Abb. 38 Metamodell für Prädikat

Es muß gefordert werden, daß Prädikatdefinitionen in Entwurfsmodellen immer im Kontext von CO-Typdefinitionen vorgenommen werden müssen.

(Constraint 45) Für alle Instanzen von ***PredicateDef*** in einem Entwurfsmodell gilt, daß die Assoziation ***contains*** definiert ist und zu einer Instanz der Metaklasse ***COTypeDef*** führt.

Zur Modellierung von prädikatbasierten Instanzmengen von COs wird im Metamodell die neue Metaklasse ***PredicateCOSetDef*** als Spezialisierung von ***COSetDef*** eingeführt (Abb. 39). Merkmal der prädikatbasierten Definition einer Instanzmenge ist die Angabe der Prädikate, die für ein CO gültig sein müssen, damit das CO zu der Instanzmenge gehört. Im Metamodell erfolgt dies durch die Spezifikation einer Assoziation zwischen

den Metaklassen ***PredicateCOSetDef*** und ***PredicateDef***. Einer Instanz von ***PredicateCODefSet*** können dabei mehrere Instanzen (mindestens eine) von ***PredicateDef*** zugeordnet werden und umgekehrt kann eine Instanz von ***PredicateDef*** zu mehreren Instanzen von ***PredicateCOSetDef*** zugeordnet sein.

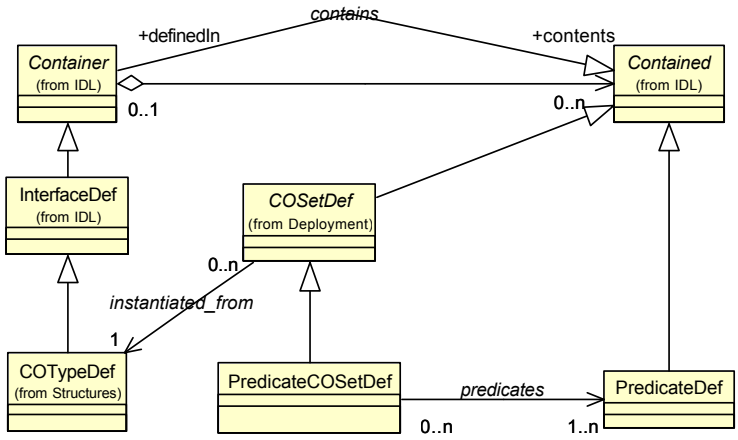


Abb. 39 Metamodell für Instanzmengen von COs basierend auf Prädikat

Es ist zu fordern, daß nur solche Prädikatdefinitionen für die Definition einer prädikatbasierten Instanzmenge von COs verwendet werden dürfen, die im Kontext des CO-Typs definiert wurden, auf dem diese Instanzmenge basiert.

(Constraint 46) Für alle Instanzen von ***PredicateCOSetDef*** in einem Entwurfsmodell gilt, daß für alle durch die Assoziation ***predicates*** zugeordneten Instanzen von ***PredicateDef*** die Assoziation ***contains*** zu derjenigen Instanz von ***COTypeDef*** führen muß, die auch durch die Assoziation ***instantiated_from*** der Instanz von ***PredicateCOSetDef*** zugeordnet wurde.

3.6.6 Bindungsfall

Mit der Einführung von prädikatbasierten Instanzmengen ist die fallbasierte Beschreibung von Güteanforderungen für Bindungen in Entwurfsmodellen möglich. Dabei soll erreicht werden, daß in einem Entwurfsmodell die Spezifikation von unterschiedlichen Bindungsfällen möglich ist, wobei jedem Bindungsfall andere Güteanforderungen zugeordnet werden können. Zur Ausführungszeit eines Softwaresystems wird der Bindungsfall durch Auswertung von Prädikaten identifiziert. Anschließend werden geeignete Maßnahmen zur Realisierung der zu dem Bindungsfall gehörigen Anforderungen durch die *Component-Support*-Plattform ergriffen. Dabei muß vorausgesetzt werden, daß eine eindeutige Identifikation des jeweils anzuwendenden Bindungsfalles zur Ausführungszeit möglich ist. Die Realisierung dieser Forderung ist in der Verantwortung des Entwicklers, da durch die Definition von Prädikaten im Entwurfsmodell noch keine Aussagen über deren spätere Auswertung möglich sind.

Zur Modellierung eines Bindungsfalles wird hier das Konzept Bindungsfall eingeführt und im Metamodell durch die Metaklasse ***BindingCaseDef*** erfaßt. Diese Metaklasse ist von der abstrakten Metaklasse ***Container*** abgeleitet.

Ein Bindungsfall enthält eine Menge von prädikatbasierten Instanzmengen von COs und eine Menge von Bindungen mit Regeln zwischen diesen Instanzmengen, denen Anforderungen als Ausdrücke über QoS-Kontrakttypen in einer geeigneten Notation hinzugefügt werden. Die Semantik von ***BindingCaseDef*** als Container wird somit wie folgt präzisiert:

(Constraint 47) Für alle Instanzen von ***BindingCaseDef*** in einem Entwurfsmodell gilt, daß die Assoziation ***contains*** ausschließlich zu Instanzen der Metaklassen ***PredicateCOSetDef*** und ***ConstrainedConnectionDef*** führt.

Zusätzlich wird gefordert, daß prädikatbasierte Instanzmengen und Bindungen mit Güteanforderungen nur innerhalb von Bindungsfällen definiert werden dürfen.

(Constraint 48) Für alle Instanzen von ***ConstrainedConnectionDef*** und ***PredicateCOSetDef*** in einem Entwurfsmodell gilt, daß die Assoziation ***contains*** definiert sein muß und zu einer Instanz der Metaklasse ***BindingCasedef*** führt.

Das Metamodell für Bindungsfälle ist in Abb. 40 dargestellt.

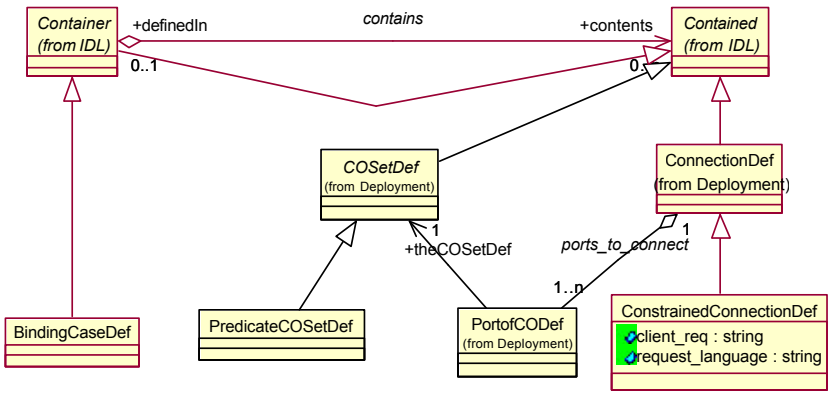


Abb. 40 Metamodell für Bindungsfall

Die zur Modellierung von Gütebeschreibungen und -anforderungen für Bindungen eingeführten Konzepte und Relationen sind im Metamodell von *CORE* innerhalb der ***Package***-Definition ***Binding*** vorgenommen worden. Aufgrund der Verwendung von Konzepten der *Deployment*-Sicht, bei der Definition der Konzepte und Relationen der Interaktionssicht ergibt sich eine Abhängigkeit im Metamodell zwischen den ***Package***-Definitionen ***Binding*** und ***Deployment***. Es besteht ebenfalls eine Abhängigkeit zu den ***Package***-Definitionen ***Structures*** und ***Configuration***, die aber bereits ausgehend von der ***Package***-Definition ***Deployment*** im Metamodell erfaßt wurde und daher nicht erneut definiert werden muß.

3.7 Modellierung des internen Verhaltens von CO-Typen

CORE realisiert keine Unterstützung für die Beschreibung des internen Verhaltens von CO-Typen. Durch die derzeit definierten Konzepte wird die Beschreibung der Struktur der Implementierung von CO-Typen durch Artefakte ermöglicht. Andererseits werden struktur- und kommunikationsbezogene Verhaltenseigenschaften durch das Entwurfsmodell festgelegt und für eine spätere automatische Erzeugung von Softwarekomponenten nutzbar gemacht. So sind Instanziierungsmuster für Artefakte und Anforderungen für Bindungen Bestandteile von Entwurfsmodellen, die mit *CORE* entwickelt werden. Auf die Definition des

internen Verhaltens, also auf die Beschreibung von Algorithmen, ist aus mehreren Gründen bewußt verzichtet worden:

- durch algorithmische Beschreibungen werden Entwurfsmodelle komplex,
- der Zeitaufwand zur Definition von Entwurfsmodellen ist gleich oder sogar höher als der zur direkten Programmierung erforderliche Zeitaufwand,
- der Lernaufwand zur Anwendung der Entwicklungstechniken steigt stark an,
- die Fehlerquote bei der Modellierung nimmt zu und erreicht das Niveau von Programmiersprachen,
- programmiersprachen eignen sich oftmals besser zur Algorithmenbeschreibung,
- mathematische Kalküle, auf denen modellbasierte Algorithmenbeschreibungen oft basieren, sind meist nur für eine spezielle Problemklasse geeignet.

Trotz der oben aufgeführten Gründe ist für einige Probleme eine modellbasierte Verhaltensbeschreibung durchaus interessant oder sogar erforderlich. Durch die objektorientierte Konstruktion von *CoRE$_{CEPT}$* ist *CoRE* auf einfache Art und Weise an diese Bedürfnisse anzupassen. So könnte beispielsweise die Metaklasse ***COTypeDef*** durch eine Metaklasse ***COTypewithBehavior*** spezialisiert werden, die dann zusätzliche Relationen zu Verhaltenskonzepten, wie z.B. Zustandsautomaten besitzen kann. *CoRE$_{TATIONS}$* sowie die Ableitungsregeln von *CoRE$_{MAP}$* sind dann natürlich ebenfalls entsprechend anzupassen. *CoRE$_{WARE}$* könnte jedoch ohne Modifikation wiederverwendet werden.

3.8 Realisierung des Metamodells

Das hier vorgestellte Metamodell zur Konstruktion von *CoRE$_{CEPT}$* wurde unter Benutzung der MOF-Meta-Metamodellelemente entwickelt. Aufgrund dieser Tatsache kann die im MOF-Standard beschriebene Technologie zur Erzeugung von CORBA-IDL-Interfacedefinitionen für den Zugriff auf Entwurfsmodelle, die Navigation in Entwurfsmodellen sowie die Manipulation von Entwurfsmodellen eingesetzt werden. Weiterhin definiert der MOF zugeordnete XMI-Standard [OMG XMI 1.1], wie eine XML-DTD erzeugt werden kann, auf deren Basis Entwurfsmodelle als XML-Dokumente repräsentiert werden.

Zum gegenwärtigen Zeitpunkt sind auf dem Markt eine Reihe von Werkzeugen verfügbar, die sich zur Ableitung der CORBA-IDL Interfacedefinitionen, einer Implementierung für diese Interfaces und eines XML-DTDs für dieses Metamodell einsetzen lassen. Allerdings akzeptieren diese Werkzeuge entweder keine UML-Beschreibung eines Metamodells als Basis für diese Ableitung [DSTC dMOF] oder die Erzeugung von Repositorien folgt nicht exakt dem MOF-Standard [Unisys UREP].

Zur Überprüfung des Metamodells von *CoRE* wurde daher durch den Autor eine eigene Realisierung des Metamodells implementiert und mit dem Verfahren zur automatischen Ableitung von Softwarekomponenten aus Entwurfsmodellen mit einem Werkzeug integriert [BK 01a]. Die Implementierung erfolgte in Form einer C++-Klassenbibliothek. Das generelle Schema der Implementierung dieser Klassenbibliothek ist, daß Metaklassen des Metamodells und ihre Attribute auf C++-Klassen mit entsprechenden *Member*-Variablen und Zugriffsmethoden abgebildet wurden. Generalisierungsrelationen im Metamodell wurden auf C++-Klassenvererbung abgebildet, konkrete Assoziationen auf Referenzen oder Listen von Referenzen, je nach Vielfachheit der Assoziationsdefinition im Metamodell. Datentypen wurden entsprechend [OMG C++ Map] auf entsprechende C++-Datentypen abgebildet. ***Package***-Definitionen im Metamodell wurden zu *Namespace*-Definitionen in der C++-Klassenbibliothek. Die *Constraint*-Definitionen wurden durch manuelle Implementierung sichergestellt. In Zukunft darf erwartet werden, daß Werkzeuge, die den MOF-Standard umsetzen, auch für die Einhaltung der *Constraint*-Definitionen in durch sie erzeugten Repositorien Sorge tragen. Dann müssen die *Constraint*-Definition aber geeignet formalisiert werden. Auf eine solche Formalisierung wurde in dieser Arbeit verzichtet, da kein Werkzeug bekannt ist, das eine automatische Behandlung von *Constraint*-Definitionen realisiert.

Dieses Schema ist unabhängig von einem konkreten Metamodell, da die oben aufgezählten Abbildungen auf der Basis der verwendeten Meta-Metamodellelemente definiert werden können. Das dann resultierende Abbildungsschema ist überblicksweise in Tab. 3 dargestellt.

MOF-Meta-Metamodellelement	C++-Konstrukt(e)
data type	Für alle Datentypen[1], die zur Definition des Metamodells verwendet wurden, werden C++-Konstrukte entsprechend [OMG C++Map] erzeugt. Dies ist möglich, da ausschließlich CORBA-IDL-Datentypen im Metamodell eingesetzt wurden.
class	Für jede Klasse wird eine C++ Klasse mit Konstruktor erzeugt.
abstract class	Für jede abstrakte Klasse wird eine C++-Klasse mit einem *Protected*-Konstruktor erzeugt.
attribute	Für jedes Attribut einer Metaklasse wird eine *Member*-Variable der C++-Klasse erzeugt. Zusätzlich wird eine *get*- und eine *set*-Methode für den Wert der *Member*-Variablen im Kontext der Klasse implementiert.
operation	Für jede Operation wird eine entsprechende Methode im Kontext der C++Klasse implementiert.
association	Für Assoziationen werden *Member*-Variablen an den beteiligten C++-Klassen erzeugt, die entsprechend der Vielfachheit der Assoziation entweder eine Referenz auf die jeweils andere Klasse als Typ haben (0,1), oder aber eine Liste mit Refernzen auf die jeweils andere Klasse (0..*, 1..*)
package	Package-Definitionen werden zu *Namespace*-Definitionen in C++, die die C++-Repräsentation aller innerhalb der *Package*-Definition enthaltenen Elemente enthält.
dependency	Abhängigkeiten werden auf *Namespace-Usage*-Konstrukte in C++ abgebildet.
constraint	*Constraint*-Definitionen sind durch entsprechenden C++-Code in den Klassendefinitionen zu realisieren, die Bereitstellung des geeigneten C++Codes erfolgt manuell.

Tab.3 Realisierung von MOF-Meta-Metamodellelementen in C++

1. Das betrifft nicht die *durch* ein Metamodell definierten Datentypen, wie z.B. ***StructDef*** in $CoRE_{CEPT}$

Ein auf dem Metamodell basierendes Entwurfsmodell läßt sich unter Verwendung der erstellten Klassenbibliothek durch Instanziierung der C++-Klassen dieser Bibliothek repräsentieren. Durch die Referenzen zwischen diesen Instanzen entsteht dabei eine Graphenstruktur (auch als Konzeptgraph bezeichnet).

Die Implementierung der Regeln für die automatische Ableitung von Softwarekomponenten konnte ebenfalls unter Verwendung der Klassenbibliothek vorgenommen werden. Dabei wird ein Entwurfsmodell als Konzeptgraph repräsentiert und anschließend wird der Graph unter Anwendung der Ableitungsregeln traversiert, wobei die Softwarekomponenten automatisch erzeugt werden.

KAPITEL 4

CoRE$_{CEPT}$ und RM-ODP

Das RM-ODP bildet die konzeptionelle Grundlage für die Konstruktion von *CoRE$_{CEPT}$*. Basiskonzepte, wie z.B. das Konzept CO-Typ, sind aus RM-ODP übernommen und in *CoRE* integriert. Das Prinzip der Strukturierung von *CoRE$_{CEPT}$* in Sichten ist ebenfalls übernommen. Allerdings stellt sich bei der kritischen Betrachtung von RM-ODP heraus, daß wichtige Konzepte dort nicht berücksichtigt sind, wohingegen andere in der Praxis nicht verwendet werden. Die Strukturierung der Sichten ist ebenfalls zu hinterfragen, da im RM-ODP das fundamentale Problem der Abbildungen und Zusammenhänge zwischen Modellen der verschiedenen Sichten nur rudimentär beschrieben wird.

An dieser Stelle sollen die eingeführten Konzepte und diejenigen des RM-ODP gegenübergestellt und diskutiert werden. Dabei werden die Konzepte der *Computational*- und der *Engineering*-Sicht aus RM-ODP herangezogen, da diese beiden Sichten durch *CoRE$_{CEPT}$* verfeinert wurden. *Enterprise*- und *Technology*-Sicht sind entsprechend der Zielstellung von *CoRE$_{CEPT}$* nicht relevant, da in *CoRE$_{CEPT}$* ausschließlich auf den Entwurf von verteilten Softwaresystemen fokussiert wird. Bezüglich der *Information*-Sicht wurde bereits ausgeführt, daß Informationsmodelle natürlich Bestandteil des Entwurfs von Softwaresystemen sind, sie dienen jedoch zur Beschreibung von Daten und stehen somit im Zusammenhang mit dem hier eingeführten Datentypmodell.

Das Datentypmodell von *CoRE$_{CEPT}$* wurde von CORBA-IDL übernommen. Es wurde ferner gezeigt, wie andere Datentypmodelle zur Beschreibung von Daten in *CoRE$_{CEPT}$* integriert werden können. Es läßt sich also festhalten, daß *CoRE$_{CEPT}$* offen für verschiedene Arten von Datentypmodellen ist und auch eine Datenmodellierung mittels der Konzepte der RM-ODP-*Information*-Sicht gestatten würde. Eine genauere Untersuchung dieses Sachverhalts soll hier nicht erfolgen, da die Erstellung von Datenmodellen nur ein Randaspekt und nicht der Schwerpunkt von *CoRE* ist.

4.1 Adaptierte Konzepte

CoRE$_{CEPT}$ baut hinsichtlich seiner Kernkonzepte auf den im RM-ODP beschriebenen Konzepten der *Computational*-Sicht auf. Einige dieser ODP-Konzepte konnten ohne Modifikation in *CoRE$_{CEPT}$* integriert werden. Dies sind solche Konzepte, die bereits in heute verfügbaren Produkten, Notationen oder Werkzeugen umgesetzt sind, deren Eignung also in der Praxis nachgewiesen wurde und als allgemein anerkannt gilt.

OPERATION, ATTRIBUT, AUSNAHME. Die Konzepte Operation, Attribut und Ausnahme sind im RM-ODP durch die *Computational*-Konzepte Operation und *Termination* enthalten. Dabei ist zu beachten, daß das Konzept Attribut sich auf das Konzept Operation zurückführen läßt, indem je nach Art des Attributs eine Lese- und Schreiboperation oder nur eine Leseoperation für den Attributwert eingeführt werden.

INTERFACETYP. Das Konzept Interfacetyp ist im RM-ODP als *Computational*-Interfacetyp enthalten und stellt dort ebenso wie in *CoRE$_{CEPT}$* ein Kernkonzept dar. Allerdings wird im RM-ODP definiert, daß ein *Computational*-Interfacetyp (dort *Interfacetemplate*[1]) jeweils nur operationale, signal- oder *stream*-basierte Interaktionselemente haben darf. Diese Restriktion wurde aus den dargestellten Gründen in *CoRE$_{CEPT}$* aufgehoben. Die durch RM-ODP gegebene Möglichkeit der Verhaltens- und Umgebungskontraktspezifikation wurde in *CoRE$_{CEPT}$* durch konkrete Konfigurationsaspekte und die Kontrakttypen konkretisiert.

CO-TYP, *SUPPORTS*- UND *REQUIRES*-RELATION. Das Konzept CO-Typ ist das grundlegende Konzept sowohl der *Computational*-Sicht von RM-ODP als auch von *CoRE$_{CEPT}$*. RM-ODP definiert ebenfalls die Möglichkeit der Instanziierung von Interfacetypen durch *Computatonal*-Objekttypen, wobei die Rolle des COs (Klient oder Server) zu beachten ist. Dieses Prinzip wurde hier durch die Relationen *requires* und *supports* umgesetzt, wobei bei einer *Requires*-Relation das CO bezüglich des Interfaces in einer Klientenrolle ist und umgekehrt bei einer supports-Relation in einer Serverrolle. Verhalten und Kontrakte von CO-Typen werden durch entsprechende Kontrakte (Kontrakttyp) an deren Interfacetypen spezifiziert. Dabei ist in *CoRE$_{CEPT}$* zusätzlich festgelegt, daß ein CO-Typ immer selbst auch ein Interfacetyp ist.

SIGNALTYP UND SIGNALPARAMETER. Signaltyp und Signalparameter sind im RM-ODP durch das *Computational*-Konzept Signal erfaßt.

CONSUME, PRODUCE, SINK, SOURCE. Die Konzepte *Consume*, *Produce*, *Sink* und *Source* stellen Interaktionselemente für signalbasierte und *stream*-basierte Interaktion dar. Im RM-ODP sind hierfür die *Computational*-Konzepte *flow* und Signal sowie die darauf aufbauenden Interfacetypen Signalinterfacetyp und *Stream*-Interfacetyp vorgesehen. Es wird durch *CoRE$_{CEPT}$* nicht *gefordert*, daß ein spezieller Signal-Interfacetyp ausschließlich signal-basierte Interaktionselemente beinhalten darf und ein spezieller *Stream*-Interfacetyp ausschließlich *stream*-basierte Interaktionselemente. Dieser Spezialfall ist aber in Entwurfsmodellen darstellbar.

ZUSTANDSATTRIBUT. Durch RM-ODP ist festgelegt, daß Objekte grundsätzlich Zustandsinformation beinhalten, die durch Aktionen oder Interaktionen verändert werden können. In *CoRE$_{CEPT}$* ist das Konzept Zustand durch das Konzept Zustandsattribut konkretisiert worden, wobei festgelegt wurde, daß Zustandsattributdefinitionen an CO-Typen erfolgen. Dieses Vorgehen setzt die Definition aus RM-ODP konsequent um.

BINDUNG MIT REGEL. Durch das RM-ODP ist definiert, daß Interfaces verbunden werden können und Bindungen die Voraussetzung für Interaktionen sind. Weiterhin ist beschrieben, daß Bindungen bestimmten Regeln genügen müssen. Dieses Prinzip ist in *CoRE$_{CEPT}$* durch das Konzept Bindung mit Regel aufgegriffen und konkretisiert worden.

1. Typ wird hier statt des mißverständlichen Terminus *Template* verwendet.

BINDUNGSFALL. Ein Bindungsfall in einem Entwurfsmodell basierend auf $CoRE_{CEPT}$ beschreibt eine Menge von Bindungen und den für sie vorgesehenen Regeln. Im RM-ODP ist festgelegt, daß *Computational*-Spezifikationen Bindungsregeln für mögliche Bindungen von CO-Typen beinhalten. Dieses Prinzip wird durch das Konzept Bindungsfall in $CoRE_{CEPT}$ konkretisiert.

4.2 Zusätzliche Konzepte

Einige Konzepte von $CoRE_{CEPT}$ haben keine direkte Entsprechung in RM-ODP, lassen sich aber als Spezialisierungen von grundlegenden RM-ODP-Konzepten und Prinzipien auf RM-ODP zurückführen. Andere sind durch RM-ODP nicht berücksichtigt, aber für die Umsetzung von *CoRE* notwendig. Diese Zusammenhänge werden im folgenden erläutert:

NAMENSRAUM. Das Konzept Namensraum ist nicht direkter Bestandteil der Konzepte der *Computational*-Sicht von RM-ODP. Im Teil 2 von RM-ODP wird jedoch ein generelles Schema für die Identifikation von beliebigen Entitäten eingeführt. Das Konzept Namensraum von $CoRE_{CEPT}$ ist eine konkrete Ausprägung dieses allgemeinen Schemas für Entwurfsmodelle, die auf $CoRE_{CEPT}$ basieren.

DATENTYP. Durch RM-ODP wird kein konkretes Datentypmodell vorgegeben. Allerdings basieren Konzepte der *Computational*-Sicht wie Parameter und Operation natürlich auf Datentypen. Die Datentypkonzepte von $CoRE_{CEPT}$ stellen eine offene und austauschbare Möglichkeit zur Beschreibung von Daten dar. Eine solche Konkretisierung ist hier notwendig, da ansonsten keine Regeln für die automatische Ableitung von Softwarekomponenten aufgestellt werden können. Solche Regeln benötigen ein wohldefiniertes Datentypmodell.

PORT-DEFINITION. Das *Port*-Konzept von $CoRE_{CEPT}$ gestattet die Konfiguration von COs und den Zugriff auf die von ihnen unterstützten Interfaces. Durch das RM-ODP ist keine explizite Konfigurationsmöglichkeit vorgegeben, es sind demzufolge auch keine entsprechenden Konzepte in der *Computational*-Sicht definiert. Allerdings definiert das RM-ODP das Prinzip von *binding actions*, die zu Bindungen zwischen COs führen. Zur Unterstützung einer automatischen Ableitung von Softwarekomponenten war eine Konkretisierung dieses Prinzips im Rahmen der Konstruktion von $CoRE_{CEPT}$ notwendig und führte zum *Port*-Konzept als Voraussetzung für den späteren Aufbau von Bindungen.

MEDIENTYP, MEDIUM, MEDIENMENGE. Die Konzepte Medientyp, Medium und Medienmenge konkretisieren das Konzept *flow* aus RM-ODP und geben diesem eine Semantik, die sich durch geeignete Ableitungsregeln und Unterstützung durch eine *Component-Support*-Plattform praktisch umsetzen läßt.

INTERAKTIONSELEMENT. Alle Interaktionselemente von $CoRE_{CEPT}$ sind innerhalb von RM-ODP definiert, die Abstraktion Interaktionselement wurde jedoch nicht vorgenommen. Dies war dort nicht notwendig, da auch das Prinzip der Kombination von verschiedenen Interaktionselementen im Kontext eines Interfacetyps im RM-ODP nicht eingeführt wurde.

ARTEFAKT, *IMPLEMENTS*-RELATION, IMPLEMENTIERUNGSELEMENT UND INSTANZIIERUNGSMUSTER. Das Konzept Artefakt ist das grundlegende Konzept zur Beschreibung der Implementierungsstruktur von CO-Typen. Durch RM-ODP wird eine derartige innere Sicht auf CO-Typen nicht definiert, die Konzentration erfolgt ausschließlich auf externe Aspekte wie Interfacetypen und Kontrakte. Dies ist ein wesentlicher Mangel des RM-ODP, da die automatische Ableitung von Softwarekomponenten ohne die Konzepte zur Beschreibung der Implementierungsstruktur nicht in ausreichendem Maße durchführbar ist. Es ließen sich dann keine Gerüste erzeugen, die durch Entwickler zum Einfügen von *Business Logic* genutzt werden können.

Ergänzt wird das Konzept Artefakt dabei von den anderen Konzepten der Implementierungssicht, wie *Implements*-Relation, Implementierungselement und Instanziierungsmuster.

SOFTWAREKOMPONENTE UND *REALIZE*-RELATION. Eine weitere wichtige Ergänzung zu den RM-ODP Konzepten stellen die Konzepte Softwarekomponente und *Realize*-Relation dar. Diese Konzepte sind durch RM-ODP nicht definiert. Dies gilt sowohl für die *Computational*- als auch für die *Engineering*-Sicht. Durch das Fehlen des Konzeptes Softwarekomponente eignet sich RM-ODP nicht zur Modellierung von komponentenbasierten Softwaresystemen. Die Bindung zwischen objektorientiertem Entwurf und komponentenbasierter Realisierung gelingt durch die ausschließliche Verwendung von RM-ODP nicht. Das Konzept Softwarekomponente ist essentiell sowohl zur automatischen Ableitung von Implementierungsgerüsten als auch zur Realisierung eines automatisierten *Deployment*-Vorganges, wie er durch *CoRE* angestrebt wird.

ASSEMBLAGE, INITIALE COS UND INITIALE BINDUNG. Assemblagedefinitionen in einem Entwurfsmodell dienen zur Beschreibung der zu einem verteilten Softwaresystem gehörigen CO-Typen, ihren initialen Instanzen sowie den initialen Bindungen. Im RM-ODP gibt es keine direkte Entsprechung für das Konzept Assemblage, es wird aber definiert, daß eine *Computational*-Spezifikation eine Beschreibung der Konfiguration von COs (also Instanzen und Bindungen) enthält. Das Konzept Assemblage ist somit die Umsetzung dieser Konfigurationsbeschreibung.

PRÄDIKAT. Das Konzept Prädikat wurde in *CoRE$_{CEPT}$* eingeführt, um Instanzmengen von CO-Typen zu definieren und ausgehend von diesen Instanzmengen eine Beschreibung der Bindungsfälle vorzunehmen. In RM-ODP ist das Konzept Prädikat definiert, es wird zur Definition von Typen von Objekten benutzt. Eine Typ ist danach eine Menge von Objekten, die einem bestimmten Prädikat genügen. Die Verwendung des Konzeptes Typ ist an dieser Stelle ungewöhnlich und führte in der Vergangenheit zur Verwirrung, da es im Widerspruch zum Typkonzept gängiger Programmiersprachen (und auch zum Typbegriff in *CoRE$_{CEPT}$*) steht. Setzt man allerdings den Typbegriff aus RM-ODP mit dem hier eingeführten Konzept einer Instanzmenge in Verbindung, so ist die Definition solcher Instanzmengen über Prädikate konzeptionell gleichzusetzten.

KONTRAKTTYP. Im RM-ODP wird das Konzept Umgebungskontrakt benutzt, um Regeln für die Interaktionen von CO-Typen mit ihrer Umgebung festzulegen. Dabei werden Güteeigenschaften als Bestandteil von Kontrakten explizit definiert. Das Konzept Kontrakttyp von *CoRE$_{CEPT}$* ist eine Umsetzung des allgemeinen RM-ODP-Konzepts von Umgebungskontrakten.

4.3 Nicht relevante ODP-Konzepte

Einige Konzepte der *Computational*- und *Engineering*-Sicht von RM-ODP wurden nicht in *CoRE$_{CEPT}$* integriert. Der Grund dafür ist, daß diese Konzepte mit der Zielsetzung einer automatischen Ableitung von Softwarekomponenten aus Entwurfsmodellen nicht benötigt werden. Diese Konzepte werden im folgenden diskutiert.

BINDING OBJECT. Nach RM-ODP sind *Binding*-Objekte spezielle Objekte, die die Bindungen zwischen anderen COs umsetzen und dabei die Einhaltung der Bindungsregeln oder Kontrakte gewährleisten. Während die Definition der Kontrakte und der Bindungen durch die Konzepte Kontrakttyp und Bindung mit Regel Bestandteil von *CoRE$_{CEPT}$* sind, wurde auf die Intergration des Konzeptes *Binding Object* verzichtet. Der Grund besteht darin, daß die Aufgaben von *Binding*-Objekten, also das Aufbauen von Bindungen und das Sicherstellen der Bindungsregeln, in der Verantwortung der *Component-Support*-Plattform liegen und daher nicht in Entwurfsmodellen erfaßt werden müssen. Ausgehend von den Entwurfsmodellinformationen über Bindungen und den für diese Bindungen geforderten QoS-Anforderungen werden durch die automatischen

Ableitungsregeln Implementierungsfragmente erzeugt, die im Zusammenwirken mit der *Component-Support*-Plattform die Funktionalität von *Binding*-Objekten realisieren.

ENGINEERING-SICHT KONZEPTE. Die Konzepte der *Engineering*-Sicht von RM-ODP sind nicht als Konzepte in $CORE_{CEPT}$ integriert worden. Die Begründung dafür ist, daß die Konzepte der *Engineering*-Sicht ausschließlich zur Modellierung einer verteilten Verarbeitungsumgebung geeignet sind, da sie allgemeine Konzepte der *Realisierung* von Objektinteraktion beinhalten. Verteilte Verarbeitungsumgebungen sind aber in Form von Komponentenarchitekturen bereits verfügbar und werden für die Umsetzung von Softwaresystemen benutzt. Sie sind aber nicht Bestandteil von Entwurfsmodellen. Vielmehr werden Spezifika der verteilten Verarbeitungsumgebung durch die Ableitungsregeln für die automatische Erzeugung von Softwarekomponenten berücksichtigt. Andere Konzepte der *Engineering*-Sicht, wie z.B. *capsule* und *cluster* sind innerhalb der *Component-Support*-Plattform umgesetzt. Eine Modellierung ist hier wiederum nicht notwendig.

Einige Konzepte der *Engineering*-Sicht sind geeignet, die Maschinen einer verteilten Umgebung für den Betrieb von Softwarekomponenten und die Eigenschaften dieser Maschinen zu modellieren. Diese Konzepte (*node*, *channel*) könnten in $CORE_{CEPT}$ aufgenommen werden, wenn zusätzlich zu der Modellierung der Softwarekomponenten auch ein Modell der Hardware vorgenommen werden soll. Ein solcher Ansatz und eine Untersuchung über die Anwendungsmöglichkeiten wird derzeit im EURESCOM Projekt P924 diskutiert. Aus der Sicht des Autors ergibt sich aber kein Anwendungsfall, der eine Erweiterung von $CORE_{CEPT}$ rechtfertigen würde. Der Grund ist, daß die Entwicklung von Softwarekomponenten und damit deren Modellierung i.allg. nicht in der Umgebung erfolgt, die für ihren Betrieb vorgesehen ist. Vielmehr ist die Ausführungsumgebung zur Entwurfszeit noch gar nicht bekannt, sie kann also auch nicht in das Entwurfsmodell aufgenommen werden. Erfolgversprechender ist der Ansatz, Anforderungen im Entwurfsmodell zu erfassen und ihre Realisierung durch die *Component-Support*-Plattform und geeignete Werkzeuge sicherzustellen.

KAPITEL 5

Notationen in CoRE$_{TATIONS}$

Die konzeptionelle Basis von *CoRE* ist durch ein Metamodell festgelegt. Darauf aufbauend werden im folgenden die in *CoRE*$_{TATIONS}$ zusammengefaßten Notationen präsentiert. Exemplarisch werden hier zwei Notationen vorgestellt - eine textuelle und eine graphische.

Ansätze, die mehrere Notationen zur Darstellung eines Modells kombinieren, sind bereits in der Vergangenheit angewendet worden [BH 98]. Dabei wurde jedoch der Zusammenhang zwischen den Teilmodellen, die mittels unterschiedlichen Notationen erstellt wurden, i.allg. durch die Definition von Ableitungsregeln zwischen jeweils zwei dieser kombinierten Notationen hergestellt. Dieses konventionelle Verfahren hat mehrere Nachteile:

- Die Realisierung der Ableitungsregeln erfolgt durch Werkzeuge, die in einem Arbeitsschritt ein Modell, das in der einen Notation repräsentiert ist, in ein Modell, das in der anderen Notation notiert ist, umsetzen. Ein iterativer Entwicklungsprozeß, wie er in der Praxis erforderlich ist, kann so nicht oder nur unvollkommen mit zusätzlichen Hilfsmitteln unterstützt werden. Bezüge zwischen Elementen, die in den verschiedenen Notationen spezifiziert sind, bleiben nur durch die Anwendung von nicht zufriedenstellenden Verfahren wie z.B. Metakommentaren erhalten.
- Die Kombination von mehr als zwei Notationen ist schwierig, da die Ableitungsregeln immer von den Konzepten einer Notation direkt auf Konzepte einer anderen definiert sind. Die Zahl der zu definierenden Regeln steigt bei der Verwendung mehrerer Notationen stark an.
- Ein Modell ist immer ausschließlich unter Verwendung der Notationen dargestellt, es gibt keine notationsunabhängige Repräsentation. Eine solche würde aber den Austausch von Modellen oder Teilmodellen zwischen verschiedenen Entwicklungsumgebungen oder die Definition und Realisierung von Ableitungsregeln auf unterschiedliche Implementierungs- oder Plattformtechnologien wesentlich erleichtern.

Der in dieser Arbeit diskutierte Ansatz *CoRE*, der auf der Trennung von Konzeptraum und Notation und im weiteren der Plattformunterstützung und der automatischen Ableitung von Softwarekomponenten beruht, ist zur Überwindung der oben aufgeführten Nachteile geeignet. Die primäre Repräsentation eines

Entwurfsmodells erfolgt immer durch Instanziierung der Konzepte von *CoRE$_{CEPT}$* für zu modellierende Entitäten, sie ist unabhängig von verwendeten Notation. Für ein Entwurfsmodell ergibt sich damit eine graphenartige Struktur, in der die Knoten gerade instanziierte Konzepte (beispielsweise ein konkreter Interfacetyp) und die Kanten instanziierte Relationen (z.B. eine konkrete Vererbungsrelation) sind. Diese Struktur wird als Konzeptgraph bezeichnet. Da der Konzeptgraph die primäre Modellrepräsentation ist, werden alle notwendigen Informationen für die Unterstützung eines iterativen Entwicklungsprozesses ebenfalls hier verwaltet. Dazu gehören z.B. Versionen von Modellelementen und Informationen über den Realisierungsstand. Bezüge zwischen in unterschiedlichen Notationen dargestellten Modellelementen müssen hingegen nicht mehr verwalten werden, da immer eine gemeinsame, primäre Repräsentation als Konzeptgraph existiert.

Werkzeuge können nun für verschiedene konkrete Notationen entweder einen Konzeptgraphen aus einem in der Notation dargestellten Entwurfsmodell erstellen oder aber in entgegengesetzter Richtung aus einem vorliegenden Konzeptgraphen ein in der gewünschten Notation repräsentiertes Entwurfsmodell erzeugen. Direkte Abbildungen zwischen zwei Notationen sind nicht mehr notwendig. Die Kombination von verschiedenen Notationen ist vereinfacht, da für eine zu integrierende Notation immer nur die Abbildung auf bzw. von einem Konzeptgraphen definiert werden muß. Um Entwurfsmodelle zwischen verschiedenen Entwicklungsumgebungen auszutauschen muß lediglich ein Austauschformat für Konzeptgraphen gefunden werden. Diese Situation ist in Abb. 41 dargestellt.

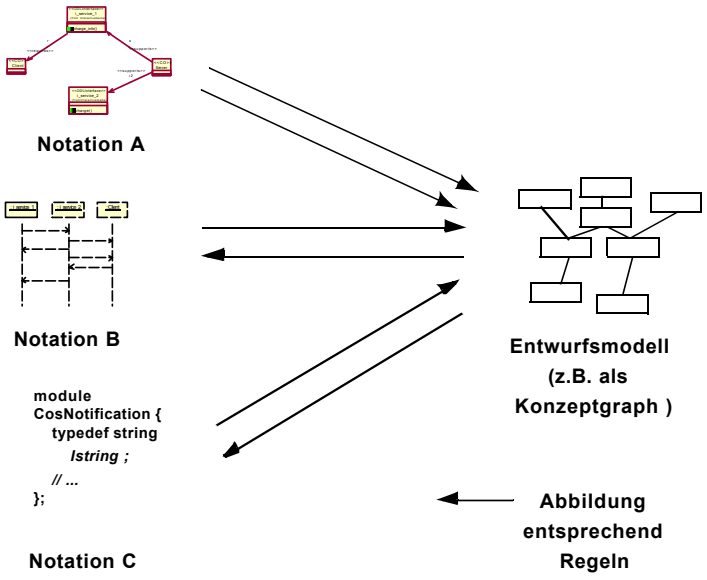


Abb. 41 Abbildung zwischen Notationen und Modellrepräsentation als Konzeptgraph

5.1 Notationsauswahl

5.1.1 Kriterien

Aus den dargestellten Sachverhalten ergibt sich, daß für die Auswahl der Notation(en) für *CoRE* ein sehr hoher Freiheitsgrad besteht. Als generelle Möglichkeiten ergeben sich neben der Definition einer neuen Notation auch die Benutzung einer vorhandenen Notation sowie der kombinierte Einsatz mehrerer Notationen. Mischformen der drei Varianten sind zusätzlich denkbar. Als Präzisierung und Ergänzung zu den in

Kapitel 1 aufgestellten Kriterien Vollständigkeit, Kompaktheit und Lesbarkeit werden für die Notationsauswahl folgende Aspekte als relevant identifiziert:

OBJEKTORIENTIERUNG. Die eingesetzte Notation sollte es gestatten, die grundlegenden Sachverhalte der Objektorientierung, insbesondere Generalisierungsrelationen, zu modellieren. Dies soll in einer kompakten Art und Weise erfolgen, also nicht durch das Auflösen von solchen Relationen durch Kopieren von Eigenschaften des Basiselements in die abgeleiteten Elemente.
Die Forderung einer objektorientierten Notation ist begründet in der Konstruktion des Konzeptraumes, die ebenfalls objektorientiert erfolgte.

GRAPHISCHE UND TEXTUELLE NOTATION. Wie bereits dargestellt ist ein wesentliches Kriterium für den Einsatz der Modellierung das einfache und u.U. sogar intuitive Verständnis von notierten Entwurfsmodellen. Damit ist die Verwendung einer graphischen Notation, trotz ihrer generellen Problematiken, anzustreben, wobei jedoch auf eine ebenso aussagekräftige textuelle Notation nicht verzichtet werden sollte. Die Austauschbarkeit beider Notationen ergibt sich dabei einzig aus der Tatsache, daß ihre Definitionen auf ein und demselben Konzeptraum beruhen.
In [CoRE I], Kapitel 1 wurde die Anforderung nach kurzen Entwicklungszeiten für Telekommunikationssoftwaresysteme aufgestellt und motiviert, daß der Aspekt der Wiederverwendbarkeit zur Realisierung dieser kurzen Entwicklungszeiten essentiell ist. Dabei ist nicht nur die Wiederverwendung der Softwarekomponenten sondern insbesondere des Entwurfsmodelles entscheidend. Wiederverwendung des Entwurfsmodells heißt aber auch *Wieder*-Verständnis der im Entwurfsmodell hinterlegten Informationen - eine graphische Repräsentation kann hierbei hilfreich sein. Andererseits sind textuell dargestellte Sachverhalte oft kompakter und können daher in Abhängigkeit von der Situation (z.B. welche Betrachtergruppe) und dem notierten Entwurfsmodell (welche Modellkonzepte wurden verwendet) leichter zu erfassen sein.

MODELLIERUNG VON STRUKTURELLEN UND VERHALTENSASPEKTEN. Entsprechend der Definition des Konzeptraumes muß mit der zu wählenden Notation sowohl die Modellierung von strukturellen Aspekten als auch die Erfassung von im Konzeptraum integrierten Verhaltensaspekten wie z.B. Instanziierungsmuster möglich sein.
Einige derzeit verwendete Notationen [OMG CORBA][W3CHTTPNG] unterstützen dagegen nur die Modellierung von Strukturaspekten - aus diesem Grund wird dieses Kriterium hier extra hervorgehoben. Es ergibt sich aus der offensichtlichen Forderung nach Vollständigkeit bezüglich der Abdeckung der Konzepte und Relationen des Konzeptraumes durch eine oder mehrere Notationen. Da der Konzeptraum sowohl strukturelle als auch Verhaltensaspekte enthält, müssen geeignete Notationen für beide gefunden werden.

TECHNOLOGIEUNABHÄNGIGKEIT. Die zu wählende(n) Notation(en) soll(en) unabhängig von einer Implementierungs- bzw. *Component-Support*-Plattform-Technologie sein. *CORE* sieht eine strikte Trennung von Konzepten, Notationen und *Component-Support*-Plattformunterstützung/Implementierung vor. Bei bisherigen Ansätzen [OMG CORBA][MS COM] ist die Notation dagegen oftmals Bestandteil einer *Component-Support*-Plattform-Technologie, was bei Verwendung der Notation auch den Einsatz der entsprechenden Technologie impliziert.
Die Anforderung der Technologieunabhängigkeit ergibt sich aus der in [CoRE I], Kapitel1 gestellten Forderung nach flexibler Adaptierbarkeit. Eine flexible Adaption von Entwurfsmodellen unabhängig von der Technologie für die Ausführungsumgebung bedingt den Verzicht auf Technologieabhängigkeiten in der gewählten Notation.

WERKZEUGUNTERSTÜTZUNG. Eine Notation ohne ausreichende Werkzeugunterstützung kann in der Praxis nicht eingesetzt werden. Werkzeugunterstützung schließt hier nicht nur die für die Notation erforderlichen Editier- und Analysewerkzeuge ein, sondern auch die Möglichkeit, einen Konzeptgraphen als unabhängige Modellrepräsentation aus einem mit der Notation spezifiziertem Entwurfsmodell erzeugen zu können. Wie

bereits diskutiert, ist eine ausreichende Werkzeugunterstützung gerade für eine graphische Notation zwingend erforderlich, um die Restriktionen bezüglich der Darstellung komplexer Entwurfsmodelle überwinden zu können.
Die grundlegende Anforderung nach Werkzeugunterstützung für den gesamten Entwicklungsprozeß und damit natürlich auch für alle zu verwendenden Notationen wurde schon in [CoRE I], Kapitel1 formuliert und begründet.

In der Praxis hat sich gezeigt, das es oft sehr schwierig ist, Akzeptanz für eine neu definierte Notation zu gewinnen. Der Grund besteht einerseits darin, daß es den Softwareentwicklern leichter fällt, bewährte Technologien einzusetzen. Andererseits ist für eine neue Notation das Kriterium Werkzeugunterstützung meist nicht oder nur mangelhaft erfüllt. Ein wesentliches Ziel dieser Arbeit ist es, *CoRE* in einem möglichst kurzen Zeitraum auch praktisch einzusetzen und damit den Nachweis der Anwendbarkeit und der Vorteile zu erbringen. Aufgrund dieser Tatsache ist hier der Verwendung von vorhandenen Notationen mit breiter Werkzeugunterstützung und deren Anpassung an die darzustellenden Konzepte der Vorzug vor einer kompletten Neuentwicklung gegeben worden.

5.1.2 UML

Die Bewertung von existierenden Notationen [OMGUML1.3][W3C HTTP NG][ITUTZ.100] anhand der aufgestellten Kriterien ergibt, daß einige der Notationen des *Unified-Modeling-Language*-Standards (UML) geeignet sind, im Rahmen von *CoRE$_{TATIONS}$* eingesetzt zu werden. Es sind aber Anpassungen vorzunehmen, um der Anforderung nach Kompaktheit und Lesbarkeit weitgehend gerecht zu werden.

Der UML-Standard vereinigt die Arbeiten der drei führenden Entwickler von objektorientierten Methoden für die Softwareentwicklung: Grady Booch, Ivar Jacobsen und James Rumbaugh. UML wurde 1997 durch *Object Management Group* (OMG) verabschiedet. Er definiert eine Reihe von hauptsächlich graphischen Notationen, die geeignet sind, verschiedene, während der Entwicklung von Softwaresystemen relevante Sachverhalte darzustellen. Im einzelnen sind dies:

USE-CASE-NOTATION. Diese Notation dient der Anforderungsanalyse und gestattet es, die wesentlichen Anwendungsfälle eines zu modellierenden Systems zu notieren. Zusammen mit den Anwendungsfällen werden auch die Akteure (Personen oder andere Softwaresysteme) spezifiziert, die in die Anwendungsfälle involviert sind. Aus *Use-Case*-Diagrammen läßt sich mithin die erwartete Systemfunktionalität entnehmen. Die *Use-Case*-Notation hat für *CoRE* keine Bedeutung, da die Phase Anforderungsdefinition als bereits abgeschlossen betrachtet wird.

KLASSEN- UND OBJEKTNOTATION. Die Klassen- und Objektnotation gestattet die graphische Modellierung der Bestandteile eines Systems (Objekte) sowie ihrer Klassifizierung (Klassen). Zusätzlich können wesentliche Relationen wie Vererbung, Assoziation oder Abhängigkeit zwischen den identifizierten Klassen spezifiziert werden. Der Klassen- und Objektnotation liegt das Prinzip der Objektorientierung zugrunde. Sie stellt die Basisnotation für viele Anwendungen des UML-Standards dar. Die Klassen- und Objektnotation wird hauptsächlich in der Entwurfs- und Implementierungsphase eingesetzt.

SEQUENZ- UND KOLLABORATIONSNOTATION. Die Sequenz- und Kollaborationsnotation erlaubt in graphischer Form die Spezifikation von Interaktionen zwischen einer Menge von Objekten. Die Sequenznotation stellt dabei den zeitlichen Verlauf in den Vordergrund, während die Kollaborationsnotation auf die Struktur und die Zusammenhänge zwischen den partizipierenden Objekten fokussiert. Beide Notationen werden zur Verhaltensspezifikation eingesetzt, sie kommen sowohl in der Phase Anforderungsdefinition als auch in den Phasen Entwurf und Implementierung zur Anwendung.

ZUSTANDSNOTATION/AKTIVITÄTENNOTATION. Die graphische Zustandsnotation wird ebenfalls zur Verhaltensspezifikation eingesetzt. Das Verhalten von Objekten wird hierbei durch die Angabe von Zustands-

automaten spezifiziert, die das ereignisorientierte Verhalten der Objekte modellieren.
Die Aktivitätennotation ist ein Spezialfall der Zustandsnotation. Sie erlaubt die Modellierung des Kontrollflusses zwischen Objekten. Beide Notationen dienen zur Modellierung des internen Verhaltens der Instanzen von Klassen. Sie werden vor allem in der Entwurfsphase eingesetzt.

KOMPONENTENNOTATION. Die Komponentennotation wird zur graphischen Notation von Softwarekomponenten eines zu modellierenden Systems, ihren Abhängigkeiten und der Zuordnung von Klassen zu den sie relasierenden Softwarekomponenten eingesetzt. Sie wird hauptsächlich in der Implementierungsphase verwendet und dient der Vorbereitung der Integrationsphase.

DEPLOYMENT-NOTATION. Die *Deployment*-Notation wird zur Modellierung der Ausführungseinheiten (z.B. Computer), die zur Ausführungszeit eines Systems vorhanden sind, ihren Abhängigkeiten und Bindungen sowie der Modellierung der Zuordnung von Komponenten zu diesen Ausführungseinheiten eingesetzt. Sie wird in der Integrationsphase sowie in der Einsatzphase verwendet.

OBJECT-CONSTRAINT-LANGUAGE-NOTATION. Die OCL-Notation ist eine Ausnahme unter den UML-Notationen, da es sich hierbei um eine textuelle Notation handelt. Sie dient der Angabe von Regeln für Modellelemente (z.B. für konkrete Klassen in einem Modell), die mit den anderen UML-Notationen dargestellt wurden. Der OCL-Notation liegt als Kalkül die Prädikatenlogik erster Stufe zugrunde. OCL wird als ergänzende Notation in allen Phasen eingesetzt.

Von den vorgestellten UML-Notationen sind für die Verwendung innerhalb von *CORE* die Klassen- und Objektnotation, die Sequenz- und Kollaborationsnotation, die Komponentennotation sowie die OCL-Notation relevant.

Die Klassen- und Objektnotation eignet sich zur Modellierung von strukturellen, Implementierungs- und Konfigurationsaspekten, die Sequenz- und Kollaborationsnotation zur Modellierung von Interaktionsaspekten und die Komponentennotation zur Modellierung von *Deployment*-Aspekten. Die OCL-Notation wird naturgemäß für die Erfassung von Regeln, wie z.B. für die Bindung von COs über ihre unterstützten bzw. angebotenen Interfaces, eingesetzt.

Die Zustandsnotation wird nicht benutzt, da $CORE_{CEPT}$ derzeit keine zustandsbasierte Verhaltensbeschreibung für CO-Typen oder Artefakte unterstützt.

Die *Deployment*-Notation wird ebenfalls nicht verwendet, da der Konzeptraum zwar das Prinzip der Erfassung von *Deployment*-Anforderungen für Komponenten unterstützt, eine Modellierung der konkreten Ausführungsumgebung selbst, wie sie mittels der *Deployment*-Notation vorgenommen werden könnte, aber nicht vorgesehen ist. Der Grund besteht darin, daß zur Entwurfszeit die Informationen über den tatsächlichen Einsatzort oftmals noch gar nicht bekannt sind.

Die UML-Notationen Klassen- und Objektnotation, Sequenz- und Kollaborationsnotation, Komponentennotation sowie die OCL-Notation erfüllen die angeführten Kriterien für die Notationsauswahl. Sie sind per Definition für die Umsetzung einer objektorientierten Modellierung geeignet, gestatten die strukturelle- und die Verhaltensmodellierung und sind bis auf die OCL-Notation graphische Notationen. Es gibt eine Reihe von Werkzeugen, die die UML-Notationen selbst unterstützen und auch weitere Standards [OMG MOF1.3] [OMG XMI1.1], die einerseits den standardisierten Zugriff auf Modelle erlauben, die unter Anwendung der UML-Notation erstellt wurden und andererseits den Austausch dieser Modelle zwischen verschiedenen Entwicklungsumgebungen vereinheitlichen.

Die UML-Notationen sind unabhängig von einer speziellen Implementierungs- oder Plattformtechnologie. Sie wurden entwickelt, um die objektorientierte Modellierung in ganz unterschiedlichen Anwendungsgebieten durch eine einheitliche graphische Darstellung zu ermöglichen. Kerngedanke der Entwicklung von UML war also die Umsetzung der objektorientierten Prinzipien und nicht die möglichst optimale Anpassung an eine konkrete Technologie.

Der Erfolg der UML-Notationen in verschiedenen Anwendungsgebieten beruht aber zu einem großen Teil auf einem hier bisher noch nicht erwähnten Aspekt: ihrer Flexibilität. Der UML-Standard definiert ein Metamodell, das alle Elemente spezifiziert, die in einem konkreten Modell instanziiert sein können, d.h. auf deren Basis Modellelemente angelegt werden. Bestandteil des Metamodells sind weiterhin Regeln, die beschreiben, auf welche Weise die Elemente zur Bildung eines Modells eingesetzt werden dürfen. So enthält das UML-Metamodell z.B. die Elemente ***class*** oder ***association*** und legt fest, daß in einem Modell Instanzen des ***class***-Elements über Instanzen des ***association***-Elements verbunden sein können. Das UML-Metamodell ist selbst wiederum unter Anwendung der UML-Klassennotation und der OCL-Notation spezifiziert worden. Der UML-Standard definiert weiterhin einen Weg, wie das UML-Metamodell und damit konkrete UML-Modelle an eine bestimmte Aufgabenstellung bzw. an ein bestimmtes Einsatzgebiet angepaßt werden können. Eine solche Anpassung wird durch Spezialisierung der Elemente des Metamodells und einer damit verbundenen Einschränkung ihrer Semantik vorgenommen. Das hat Auswirkungen auf die Modelle, die unter Verwendung solcher, durch Spezialisierung angepaßter Metamodellelemente erzeugt werden können. Mit diesem Mechanismus ist es jedoch nicht möglich, neue Basiselemente zum UML-Metamodell hinzuzufügen oder die Semantik der im UML-Metamodell enthaltenen Elemente selbst zu verändern. Jegliche neue Semantik muß durch die Angabe von Regeln für die definierten Spezialisierungen formuliert werden.

Die Spezifikation einer solchen Metamodellerweiterung ist ein UML-Profil. OMG macht von der Möglichkeit, UML-Profile zu spezifizieren und damit den UML-Standard an unterschiedliche Anwendungsgebiete anzupassen, derzeit in verschiedenen Standardisierungsaktivitäten Gebrauch [OMG CORBA P].

Die Erweiterung des UML-Metamodells erfolgt unter Verwendung der drei folgenden Mechanismen:

- *Stereotype*-Definition
  Stereotype-Definitionen (Stereotypen) spezifizieren die Spezialisierungen von Elementen des UML-Metamodells. Diese Spezialisierungen können zusätzliche Attribute sowie eine eingeschränkte Semantik ihrer Verwendung in einem konkreten Modell beinhalten. Stereotypen dürfen selbst wiederum durch andere Stereotypen spezialisiert werden.
- *Tagged-Value*-Definition
  Zu Modellelementen können beliebige Informationen in Form von Eigenschaftslisten hinzugefügt werden. Die Elemente solcher Listen sind Paare von Marken (*tags*) und Werten. Eine Marke ist eine Zeichenkette, die eindeutig innerhalb der jeweiligen Eigenschaftsliste ist. Für Werte wird der Typ Zeichenkette zwar nicht vorgeschrieben, aber im Interesse einer Austauschbarkeit von Modellen ist dieses ebenfalls üblich. Für einen gegebenen Stereotypen kann nun eine Liste von Marken mit möglicherweise vorhandenen *Default*-Werten spezifiziert werden. Damit erhält die *Stereotype*-Definition Attribute, zusätzlich zu den Attributen ihres Basis-Metamodellelements.
  Das Konzept der *Tagged-Value*-Definitionen kann direkt auf Elemente im Modell angewendet werden, ohne zuvor eine *Stereotype*-Definition vorgenommen zu haben. Soll aber ausgedrückt werden, daß alle Modellelemente, die für ein bestimmtes Metamodellelement angelegt werden, einen definierten Satz von zusätzlichen Attributen besitzen sollen, so muß eine *Stereotype*-Definition mit der Angabe einer zusätzlichen *Tagged-Value*-Liste erfolgen, die die gewünschten Marken und ggf. ihre *Default*-Werte spezifiziert.
- *Constraint*-Definition
  Durch *Constraint*-Definitionen innerhalb eines UML-Profiles wird die Verwendung der UML-Metamodellelemente und -relationen zur Spezifikation von Modellen, die dem UML-Profil genügen, eingeschränkt. Dabei wird insbesondere die Verwendung der durch *Stereotype*-Definitionen eingeführten Spezialisierungen der UML-Metamodellelemente präzisiert. Die Einhaltung der *Constraint*-Definitionen führt zu Modellen, die bezüglich des UML-Profils korrekt formuliert sind.

Die Definition von *Stereotype*-Definitionen, *Tagged-Value*-Definitionen und *Constraint*-Angaben eines Profiles kann mittels der UML-Notation erfolgen, da auch das UML-Metamodell selbst in dieser Notation vorliegt. Werkzeuge, die die UML-Notationen unterstützen, sind damit in der Lage, derart vorgenommene Metamodellerweiterungen zu lesen und damit die Spezifikation von Modellen bezüglich dieser Erweiterungen zu gestatten.

Der UML-Erweiterungsmechanismus sichert also nicht nur eine hohe Flexibilität der Notationen durch Anpassung an spezifische Einsatzgebiete. Es ist außerdem möglich, bestehende Werkzeuge für die Spezifikation von Modellen für die Erweiterungen einzusetzen und den Austauch von Modellen zwischen solchen Werkzeugen auch nach der Verwendung von Erweiterungen zu erlauben.

In Abschnitt 5.2 wird ein UML-Profil zur Darstellung der Konzepte von *CoRE*$_{CEPT}$ als Bestandteil von *CoRE*$_{TATIONS}$ entwickelt. Dieses UML-Profil wird im Rahmen von *CoRE* als Hauptnotation angesehen. Die Mehrzahl der Entwurfsmodelle, die mit *CoRE* erstellt werden, werden unter Benutzung dieses Profils notiert.

5.1.3 Eine textuelle Syntax

Durch die ausschließliche Verwendung von graphischen Darstellungsmitteln kann die Präsentation von Entwurfsmodellen eine hohe Komplexität erreichen, so daß die Kriterien Lesbarkeit und Kompaktheit nicht mehr erfüllt werden können. Darüber hinaus zeigt die Erfahrung des Autors, daß ein Teil der potentiellen Anwender von *CoRE* eine textuelle Notation von Entwurfsmodellen bevorzugt. Ein Beispiel für diese Präferenz ist die Spezifikation von CORBA-Interfacedefinitionen: Mit dem UML-Profil für CORBA-IDL ist eine graphische Darstellung von Spezifikationen (CORBA-Modelle) möglich. Weit verbreitete UML-Werkzeuge unterstützen diese Notation bis hin zur automatischen Verarbeitung der resultierenden Definitionen. Trotzdem verwenden CORBA-Entwickler häufig die textuelle Notation CORBA-IDL, die eine kompakte Darstellung von CORBA-Modellen gestattet. Diese kompakte Darstellung wird insbesondere im CORBA-Standard selbst benötigt.

CoRE ermöglicht die Nutzung verschiedener Notationen zur Darstellung von Entwurfsmodellen. Die Semantik der notierten Modellelemente ist durch das Metamodell fixiert, so daß ausschließlich die bidirektionale Abbildung zwischen Elementen des Metamodells und Notationskonstrukten erklärt werden muß. Konsequenterweise wird *CoRE*$_{TATIONS}$ um die Definition einer textuellen Notation ergänzt.

In [CoRE I], Abschnitt 3.1 wurde diskutiert, daß der Ausgangspunkt bei der Definition von *CoRE*$_{CEPT}$ die Konzepte der Beschreibungssprache ITU-ODL [ITUTZ.130] waren. In ihrer gegenwärtig standardisierten Form umfaßt diese Beschreibungssprache Konzepte der eingeführten Struktursicht von *CoRE*$_{CEPT}$. Insbesondere gestattet ITU-ODL die Notation von CO-Typen und durch diese angebotene bzw. genutzte Interfacetypen. Durch die Fundierung von ITU-ODL auf CORBA-IDL übernimmt diese Beschreibungssprache auch das Datentypmodell von CORBA-IDL. Die Spezifikation von Interfacetypen in ITU-ODL ist für operationale und *Contiuous-Media*-Interaktionsarten möglich, die Beschreibung von Signalinteraktionselementen an Interfaces ist in der aktuellen Version nicht vorgesehen. Im Gegensatz zu den hier vorgestellten Konzepten werden Interfacetypen in ITU-ODL nach Interaktionsarten unterschieden, es existieren also spezifische Interfacetypen für operationale und *Continuous-Media*-Interaktionsarten.

Die konzeptionelle Nähe der Darstellungsmittel von ITU-ODL zu den Konzepten von *CoRE*$_{CEPT}$ legt natürlich nahe, die zu erstellende textuelle Notation auch tatsächlich unter Verwendung dieser Darstellungsmittel zu definieren. Ein zusätzlicher Vorteil dieser Variante besteht darin, daß existierende ITU-ODL- und CORBA-IDL-Spezifikationen ohne Modifikationen *CoRE*-Entwurfsmodelle sind.

Eine Integration der bestehenden Konzepte von ITU-ODL mit denjenigen von *CoRE* impliziert natürlich eine Harmonisierung des Konzeptes Interfacetyp in ITU-ODL mit der hier definierten Semantik eines Interfacetyps als Kontext für Interaktionselemente potentiell unterschiedlicher Interaktionsarten. Eine derartige Harmonisierung leistet einen wesentlichen Beitrag zur Fundierung der *Continuous-Media*-Interfaces in ITU-ODL, die bisher versäumt wurde. Daneben trägt die Erweiterung dieser Beschreibungssprache um Konzepte der hier definierten Konfigurations- und Implementierungssichten dazu bei, ITU-ODL zur Beschreibung der Komponenten konkreter Softwaresysteme weiterzuentwickeln.

Die auf ITU-ODL basierende textuelle Beschreibungssprache wird als extended ODL (*eODL)* bezeichnet und in *CoRE*$_{TATIONS}$ aufgenommen. Sie wird in Abschnitt 5.3 definiert.

5.2 UML-Profildefinition für CoRE$_{TATIONS}$

Im folgenden wird die Definition eines UML-Profiles zur Notation von Entwurfsmodellen vorgenommen, die auf dem Metamodell von *CoRE* basieren. Es sind die UML-Metamodellelemente und die in dem zu entwickelnden Profil vorgenommenen Spezialisierungen von UML-Metamodellelementen anzugeben, die verwendet werden müssen, um ein auf dem Metamodell basierendes Entwurfsmodell zu notieren. In diesem Sinne entstehen Ableitungsregeln, die definieren, welche Instanzen von *CoRE*-Metamodellelementen durch ein UML-Modell ausgedrückt werden und umgekehrt, wie eine Menge von *CoRE*-Metamodellinstanzen in Form eines UML-Modells notiert werden kann.

5.2.1 Regeln für UML-Profildefinition

Eine UML-Profildefinition erfolgt durch die Spezifikation von *Stereotype*-Definitionen, *Tagged-Value*-Definitionen sowie den zugehörigen *Constraint*-Angaben. Es gibt allerdings derzeit noch keine standardisierten Vorgehensweise, wie diese Spezifikation im Detail zu erstellen ist. Hinweise und Vorschläge finden sich allerdings in [OMG CORBA]. Danach sollen in einer Profildefinition:

- eine (für die Aufgabenstellung relevante) Teilmenge des UML-Metamodells identifiziert werden,
- neue allgemeine Modellelemente für alle Modelle (Instanzen der in der identifizierten Teilmenge enthaltenen Elemente sowie der *Stereotype*-Definitionen) spezifiziert werden,
- neue UML-Modellkonstrukte[1] (über *Sereotype*- und *Tagged-Value*-Definitionen) als Spezialisierungen von denen der identifizierten Teilmenge des UML-Metamodells definiert werden,
- die Semantik der neuen Modellkonstrukte spezifiziert werden, die über die Semantik der identifizierten Teilmenge des UML-Metamodells hinausgeht sowie,
- *Constraint*-Definitionen vorgenommen werden, die über diejenigen des UML-Metamodells selbst hinausgehen und die Definitionen der UML-Metamodellelemente und der neu definierten Modellkonstrukte ergänzen.

Die Definition des UML-Profiles für *CoRE* folgt grundsätzlich diesen Regeln - allerdings sind Definitionen von *Constraint*s für die Verwendung von Modellkonstrukten des Profiles zur Beschreibung von Modellen nicht erforderlich. Der Grund hierfür besteht darin, daß in *CoRE$_{TATIONS}$* zusätzlich eine Relation zu dem Metamodell von *CoRE* in Form von Ableitungsregeln definiert wird. Ein zum UML-Profil von *CoRE* konformes UML-Modell ist dann korrekt formuliert, wenn die Anwendung der Regeln zum Aufbau eines auf dem Metamodell basierenden Entwurfsmodells zu einem Entwurfsmodell führt, für das die in Kapitel 3 aufgestellten *Constraint*-Definitionen gelten. Eine Angabe von *Constraint*-Definitionen für das UML-Profil ist redundant.

Der Verzicht auf *Constraint*-Definitionen wurde möglich, da ein von dem UML-Metamodell unabhängiges Metamodell von *CoRE* existiert. Anderenfalls würde das UML-Metamodell mit den im Profil definierten Erweiterungen die Rolle des Metamodells von *CoRE* einnehmen, wobei dann die *Constraint*-Definitionen des UML-Metamodells zusammen mit den im Profil vorgenommenen Ergänzungen die korrekte Formulierung von Entwurfsmodellen sicherstellen würden.

Die Entscheidung zugunsten eines eigenständigen, UML-unabhängigen Metamodells von *CoRE* ist getroffen worden, da ansonsten eine zu starke Technologieabhängigkeit von *CoRE* (in Bezug auf die Notation) und eine Fixierung von UML als der primär einzusetzenden Notation gegeben ist. Außerdem ist nur eine Teilmenge von UML-Konstrukten und damit des UML-Metamodells notwendig, um Entwurfsmodelle entsprechend den Konzepten von *CoRE$_{CEPT}$* formulieren zu können. Ein Metamodell unter Verwendung des

1. UML-Modellkonstrukt bedeutet im folgenden UML-Metamodellelemente sowie auf diesen basierende *Stereotype*-Definitionen im UML-Profil

UML-Metamodells hätte somit einen erheblichen Mehraufwand (durch impliziten Einschluß von nicht benötigten Konstrukten) bedeutet und die Modellierung des Metamodells nicht adäquat gestattet.

5.2.2 Relevante Teilmenge des UML-Metamodells

Die relevante Teilmenge des UML-Metamodells ergibt sich durch die zur Notation der Konzepte von *CoRE$_{CEPT}$* im folgenden definierten *Stereotype*-Definitionen sowie die direkt zur Notation eingesetzten UML-Metamodellelemente. Jede *Stereotype*-Definition basiert dabei wiederum auf einem UML-Metamodellelement. Implizit sind natürlich auch alle Basiselemente der verwendeten UML-Metamodellelemente Bestandteil der relevanten Teilmenge des UML-Metamodells für das hier definierte UML-Profil, diese werden aber nachfolgend nicht explizit aufgeführt.

Die Teilmenge des UML-Metamodells, die im Rahmen der Definition des UML-Profils Verwendung findet, besteht aus den in Tab. 4 enthaltenen Elementen.

UML-Metamodellelement	UML-Metamodell-Paket	verwendet für
Association	***CoRE***	Signalparameter, *Consume, Produce, Sink, Source, Supports, Requires, Provided-Port, Used-Port, Implements*
Attribute	***CoRE***	Attribut, Zustandsattribut, Prädikat, Dimension (QoS-*Contract*-Typ)
Class	***CoRE***	Strukturierte Datentypen, Interfacetyp, Signaltyp, Medientyp, Medium, Medienmenge, CO-Typ, Artifakt, QoS-Kontrakttyp
Constraint	***CoRE***	Bindung mit Regel
Dependency	***CoRE***	Softwarekomponente
Operation	***CoRE***	Operation, Implementierungselement
Parameter	***CoRE***	Operation
DataType	***CoRE***	Grunddatentypen
Exception	***Common Behavior***	Ausnahme
Package	***Model Management***	Namensraum
Subsystem	***Model Management***	Assemblage
TaggedValue	***Extension Mechanisms***	Instanziierungsmuster, Implementierungselement, Eigenschaft
Abstraction	***CoRE***	Implementierungselement
Component	***CoRE***	Softwarekomponente, *Realize*
Collaboration	***Collaboration***	Initiale COs, Initiale Bindung, Bindung mit Regel, Bindungsfall
ClassRole	***Collaboration***	Initiale COs

Tab. 4 Relevante Teilmenge des UML-Metamodells für das UML-Profil

UML-Metamodellelement	UML-Metamodell-Paket	verwendet für
AssociationRole	***Collaboration***	Initiale Bindung, Bindung mit Regel
AssociationEnd	***CoRE***	Attribut
Binding	***CoRE***	Strukturierte Datentypen
Comment	***CoRE***	Kommentar
ElementImport	***Model Management***	Namensraum
ElementOwnership	***CoRE***	Namensraum
Permission	***CoRE***	Namensraum
Usage	***CoRE***	Namensraum

Tab.4 Relevante Teilmenge des UML-Metamodells für das UML-Profil

5.2.3 Allgemeine Modellelemente

Allgemeine Modellelemente sind diejenigen Modellelemente, die grundsätzlich in jedem Modell vorhanden sind und so für die Beschreibung anderer Modellelemente zur Verfügung stehen. Kandidaten für allgemeine Modellelemente sind beispielsweise Grunddatentypen oder Basiselemente für andere Arten von Modellelementen wie z.B. CO-Typen und Interfacetypen.

Im Metamodell von *CoRE* sind Grunddatentypen wie ***boolean*** oder ***short*** als Primitive modelliert worden. Diese Primitive werden durch das UML-Profil als allgemeine Modellelemente festgelegt und als Klassen spezifiziert. Zu ihrer Kennzeichnung wird eine *Stereotype*-Definition ***BaseType*** vorgenommen, die von dem UML-Metamodellelement ***Class*** abgeleitet ist. Zusätzlich zu den Primitiven werden noch ***ValueBase*** und ***TypeCode*** als allgemeine Modellelemente in das UML-Profil aufgenommen. ***ValueBase*** dient als implizite Basisklasse für alle *Value*-Typdefinitionen und ***TypeCode*** wird aus Kompatibilitätsgründen zu CORBA-IDL eingeführt, da das Metamodell von CORBA-IDL wiederum Basis für das in *CoRE* definierte Metamodell ist.

In Abb.42 sind alle allgemeinen Modellelemente des UML-Profils notiert.

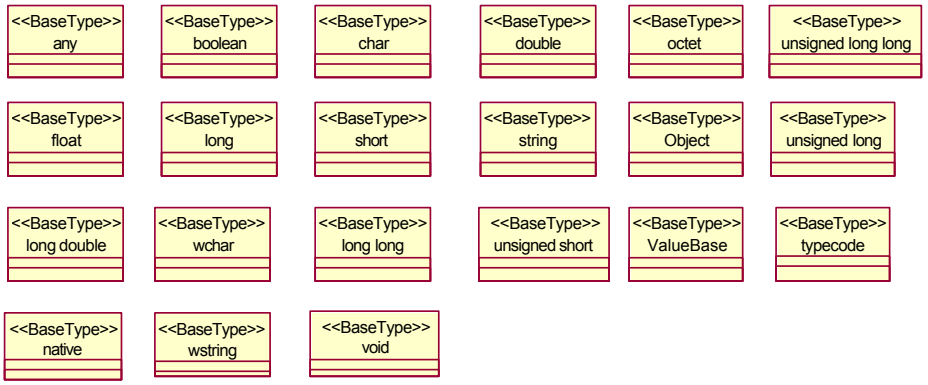


Abb. 42 Allgemeine Modellelemente

Die Definition von allgemeinen Modellelementen durch das UML-Profil soll sich auf ein Minimum beschränken und nur diejenigen Elemente beinhalten, die tatsächlich universelle Bedeutung in allen Entwurfsmodellen haben. Andere Elemente, die nur für einzelne Entwurfsmodelle oder bestimmte Mengen von Entwurfsmodellen als Basiselemente benutzt werden, können durch Pakete (wiederum Entwurfsmodelle) bereitgestellt werden. Kandidaten hierfür sind spezielle vordefinierte Medientypen wie MPEG2 oder MP3 [RFC MIME]. Da diese aber nur Verwendung finden, wenn das zu modellierende Softwaresystem die Interaktionsart *Continous-Media* benötigt, werden sie nicht als allgemeine Modellelemente durch das UML-Profil definiert. Hersteller von Hardware für *Continous-Media* könnten zusammen mit der Hardware auch ein Entwurfsmodell ausliefern, das alle von der Hardware unterstützten Medientypen enthält und zusätzlich vorgefertigte Artefakte, die die Verarbeitung von auf diesen Medientypen basierenden Medien in *Continous-Media*-Interaktionen übernehmen. Eine solche Vorgehensweise ist die konsequente Erweiterung des heute üblichen Verfahrens der Bereitstellung von Bibliotheken und *Application Programming Interfaces* (APIs) bezüglich einer modellbasierten Entwicklung.

5.2.4 Zusätzliche Modellkonstrukte und ihre Semantik

Im folgenden werden die zur Notation von Entwurfsmodellen, die auf dem Metamodell von *CoRE* basieren, benötigten zusätzlichen Modellkonstrukte von UML im Rahmen des UML-Profils als Spezialisierungen von UML-Metamodellelementen eingeführt. Die Definition ihrer Semantik erfolgt durch Angabe der Relation zwischen diesen Konstrukten und den Elementen des Metamodells von *CoRE*. Deren Semantik wiederum ist in Kapitel3 beschrieben.

Zur Notation der Spezialisierungen der UML-Modellkonstrukte wird hier UML eingesetzt. Dies ergibt sich aus der Tatsache, daß das UML-Metamodell selbst mittels der UML-Notation beschrieben ist. Das dadurch definierte UML-Modell wird auch als virtuelles Metamodell von *CoRE$_{TATIONS}$* (in Anlehnung an den Begriff „*Virtual Meta Model*“ aus [OMG CORBA P]) bezeichnet.

Stereotype-Definitionen werden im virtuellen Metamodell des UML-Profils als UML-Klassen beschrieben, die mit dem Stereotyp **<<stereotype>>** gekennzeichnet sind. Das Basiselement des UML-Metamodells, das durch eine *Stereotype*-Definition spezialisiert wird, ist durch eine **Dependency**-Relation mit dem Stereotyp **<<baseElement>>** mit der *Stereotype*-Definition verbunden.

Entsprechend der Semantik des UML-Erweiterungsmechanismus sind Spezialisierungen von *Stereotype*-Definitionen durch andere *Stereotype*-Definitionen möglich, dies wird notiert durch eine Generalisierungsrelation zwischen den UML-Klassen den beteiligten *Stereotype*-Definitionen. Es ist zudem möglich, *Stereotype*-Definitionen als abstrakt zu erklären, mit der Semantik, daß in einem Entwurfsmodell keine Instanzen mit diesem Stereotyp angelegt werden dürfen. Eine abstrakte *Stereotype*-Definition wird durch eine abstrakte UML-Klasse mit dem Stereotyp **<<stereotype>>** notiert. Assoziationen zwischen *Stereotype*-Definitionen sind allerdings nicht möglich, da das UML-Metamodell *Stereotype*-Definitionen nicht als **Classifier**-Element erklärt und Assoziationen immer zwischen **Classifier**-Elementen definiert werden müssen. Jedoch können **Dependency**-Relationen in dieser Situation durchaus erklärt werden.

Zur Notation von *Tagged-Value*-Definitionen im Kontext von *Stereotype*-Definitionen werden UML-Attributdefinitionen verwendet, wobei das Attribut der für die *Stereotype*-Definition eingeführten UML-Klasse zugeordnet wird. Die UML-Notation für die erläuterten Elemente des virtuellen Metamodells ist in Abb. 43 dargestellt. Das Konzept **Package** wird analog zur Definition des Metamodells von *CoRE* auch innerhalb der Definition des virtuellen Metamodells zu Strukturierungszwecken verwendet.

Analog zu der Definition des Metamodells von *CoRE* lassen sich auch für die Realisierung des UML-Profils bereits existierende Arbeiten nutzen. So existiert bereits ein standardisiertes UML-Profil, welches die Notation von CORBA-IDL-Spezifikationen mit UML erlaubt. Da das Metamodell von *CoRE* auf der Basis des CORBA-IDL-Metamodels entwickelt wurde, ist es folgerichtig, das UML-Profil von *CoRE$_{TATIONS}$* auf dem UML-Profil für CORBA-IDL [OMG CORBA P] basieren zu lassen.

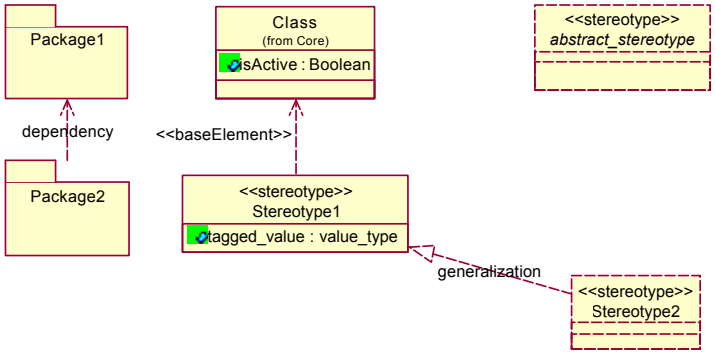


Abb. 43 UML-Notation des virtuellen Metamodells

Alle im folgenden eingeführten Modellkonstrukte werden in einem UML-Modell definiert, daß die in Abb. 44 dargestellte Struktur in Form von ***Package***-Definitionen besitzt. Diese Struktur orientiert sich dabei an der ***Package***-Struktur des Metamodells von *CoRE*. Die ***Package***-Definitionen ***Foundation***, ***Behavioral Elements*** und ***Model Management*** beinhalten den Teil des UML-Metamodells, der für die Definition des UML-Profils verwendet wird. ***CORBAProfil*** enthält die Konstrukte des UML-Profiles für CORBA-IDL, die anderen ***Package***-Definitionen beinhalten die zusätzlich benötigten Konstrukte zur Notation von Entwurfsmodellen entsprechend *CoRE*.

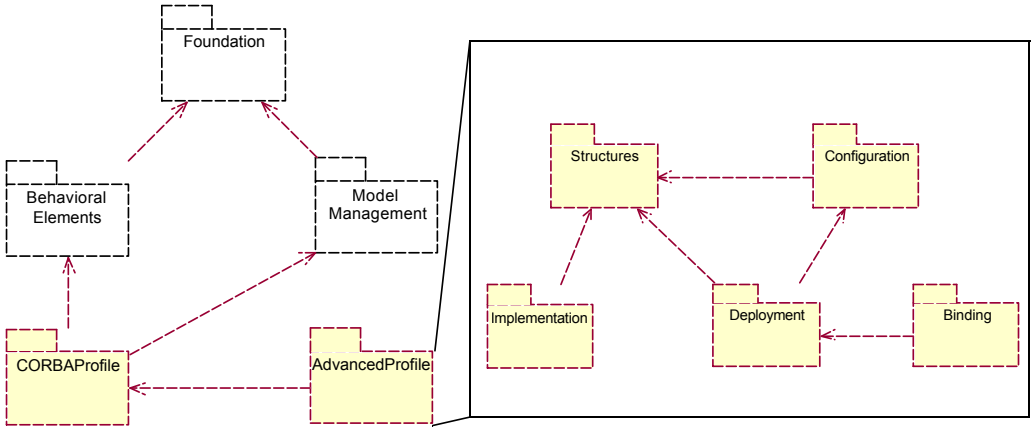


Abb. 44 Struktur des UML-Profils

An einigen Stellen wurde von der in [OMG CORBA P] beschriebenen Definition des UML-Profiles für CORBA-IDL abgewichen. Diese Abweichungen betreffen insbesondere die Bezeichner von *Stereotype*-Definitionen. Diese Abweichungen werden in den nachfolgenden Abschnitten erläutert. Sie sind vorgenommen worden, um in der Darstellung eines Entwurfsmodells eine unnötige und unbeabsichtigte Anbindung an CORBA-Technologie zu vermeiden. Die Darstellung eines Entwurfsmodells mit dem UML-Profile impliziert nämlich nicht die Realisierung des modellierten Systems mit CORBA-Technologien. Ist aber in den Bezeichnern die Zeichenkette „***CORBA***“ enthalten, so könnte ein Anwender eine solche Abhängigkeit annehmen.

5.2.4.1 Ableitungsmuster für UML-Profile aus Metamodellen

Die Definition eines UML-Profils zur Notation von Entwurfsmodellen bezüglich wohl-definierter Metamodelle ist kanonisch in dem Sinne, daß sich allgemeine Ableitungsmuster definieren lassen, mit denen festgelegt wird, welche UML-Konstrukte zur Notation von Modellelementen benutzt werden. Diese Ableitungsmuster basieren auf dem Meta-Metamodellelement, das dem jeweiligen Metamodellelement eines darzustellenden Modellelements zugrunde liegt.

Natürlich können die Ableitungsmuster die Semantik der Elemente eines Metamodells nicht berücksichtigen, aus diesem Grunde können keine *automatischen* Ableitungsregeln für UML-Profile aus Metamodellen gefunden werden. Die Ableitungsmuster lassen daher Alternativen für die Definition der Notation von Entwurfsmodellen zu. Die folgende Tabelle gibt einen Überblick der verwendeten Ableitungsmuster.

Meta-Metamodellelement	UML-Profil-Konstrukt
Class	• Standard-UML-Metamodellelement, das auf ***Classifier***, ***Relationship*** oder ***BehavioralFeature*** basiert. • *Stereotype*-Definition im Profil basierend auf dem UML-Metamodellelementen ***Classifier***, ***Relationship*** oder ***BehavioralFeature*** mit dem Namen der Metaklasse des Metamodells als *Stereotype*-Namen. Kriterium: Ist ein UML-Metamodellelement vorhanden, das die geforderte Semantik exakt bereitstellt (UML-Semantik entspricht Konzeptsemantik), dann wird dieses zur Notation eingesetzt. Wird die Konzeptsemantik nicht exakt durch das UML-Metamodellelement bereitgestellt, dann wird eine *Stereotype*-Definition im UML-Profil vorgenommen. Besteht die Konzeptsemantik darin, andere Modellelemente zu beinhalten, dann basiert die *Stereotype*-Definition auf ***Classifier***. Soll hingegen ein Zusammenhang hergestellt werden, dann basiert die *Stereotype*-Definition auf ***Relationship***. Ist ein Verhaltenskonzept zu notieren, dann basiert die *Stereotype*-Definition auf ***BehavioralFeature***.
AbstractClass	Für ***AbstractClass*** gelten dieselben Kriterien und Auswahlmöglichkeiten, wie für ***Class***, allerdings ist das zur Notation eingesetzte Konstrukt abstrakt, d.h. entweder eine abstrakte Klasse im UML-Metamodell oder eine abstrakte *Stereotype*-Definition.
Attribute	Im UML-Metamodell definiertes Attribut an einer UML-Metamodellklasse. *Tagged-Value*-Definition mit dem Namen des Attributes als Namen und dem Typ des Attributes als Typ an einem UML-Metamodellelement oder einer *Stereotype*-Definition. Im UML-Metamodell definierte Assoziation, wenn der Typ des Attributes eine Metaklasse ist.
Operation	nicht anwendbar
Association	Standard-Assoziationsdefinition aus dem UML-Metamodell. *Stereotype*-Definition basierend auf UML-Metamodellelement Association.
Aggregation	analog zu ***Association***
Generalization	Generalisierungsrelation zwischen *Stereotype*-Definitionen
DataType	***PrimitiveType*** *Stereotype*-Definition basierend auf dem UML-Metamodellement ***Classifier***
Package	***Package***-Definition im UML-Modell des UML-Profils
Constraint	nicht anwendbar

Tab.5 Ableitungsmuster für UML-Profile aus Metamodellen

5.2.5 Struktursicht

Strukturelle Informationen über Softwaresysteme werden in UML generell mittels der Klassennotation notiert. Darstellungsmittel der Klassennotation sind Klassen, deren Operationen und Attribute sowie ihre Relationen. Diese Relationen können verschiedene Semantik besitzen, wie allgemeine Relation (Assoziation), Generalisierung oder Abhängigkeit.

Zur Notation der Konzepte der Struktursicht werden diese Darstellungsmittel eingesetzt, wobei geeignete *Stereotype*-Definitionen die Zuordnung der Darstellungsmittel zu Konzepten des Konzeptraumes und damit letztlich die Abbildung von UML-Modellen auf Entwurfsmodelle von *CoRE* und umgekehrt ermöglichen.

5.2.5.1 Namensraum

Das Konzept Namensraum wurde im Metamodell von *CoRE* durch eine Metaklasse ***NamespaceDef*** modelliert, die von der Metaklasse ***ModuleDef*** aus dem CORBA-IDL-Metamodell abgeleitet ist. In UML wird das Konzept von Namensräumen durch ***Package***-Definitionen mit darin enthaltenen anderen Modellelementen dargestellt.

Dieses Konzept wird auch im UML-Profil von *CoRE$_{TATIONS}$* aufgegriffen, wobei zur Kennzeichnung die *Stereotype*-Definition ***Namespace*** als Spezialisierung der im UML-Profil für CORBA-IDL enthaltenen *Stereotype*-Definition ***CORBAModule*** eingeführt wird. ***CORBAModule*** wird außerdem in Module umbenannt. Diese *Stereotype*-Definition basiert auf dem UML-Metamodellelement ***Package***.

Durch die Definition in unterschiedlichen UML-***Package***-Definitionen des virtuellen Metamodells führt die Namensgleichheit zwischen dem hier eingeführten Stereotyp ***Namespace*** und der bereits im UML-Metamodell enthaltenen gleichnamigen Klasse nicht zu einem Konflikt. Das virtuelle Metamodell für Namensraum ist in Abb.45 dargestellt. Da für das Konzept Namensraum im Metamodell keine Attribute definiert sind, erhält auch die *Stereotype*-Definition im virtuellen Metamodell keine Attribute.

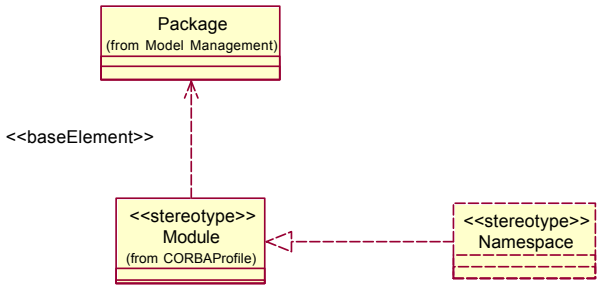


Abb. 45 Virtuelles Metamodell für Namensraum

Es ist in Realisierungen des UML-Profils durch Werkzeuge erlaubt, auf die Angabe des Stereotyps ***Namespace*** zu verzichten, wenn die Auswahl der ***Package***-Definitionen für die Erzeugung des Entwurfsmodells eindeutig ist, d.h. wenn von allen ***Package***-Definitionen im UML-Modell bekannt ist, daß sie Namensräume im Sinne von *CoRE$_{CEPT}$* darstellen.

(Beispiel 6) Wenn ***M2*** als Namensraum in einem Namensraum ***M1*** eines Entwurfsmodells definiert ist, so wird diese Situation in UML wie in Abb. 46 dargestellt.

5.2.5.2 Datentyp, Interfacetyp, Operation, Attribut und Ausnahme

Zur Darstellung der Konzepte Datentyp, Interfacetyp (ausschließlich operationale Interaktionsart), Operation, Attribut und Ausnahme wird hier die durch das UML-Profil für CORBA-IDL festgelegte Notation eingesetzt. Es wird allerdings von allen dort eingeführten *Stereotype*-Definitionen der Präfix ***CORBA*** aus den oben

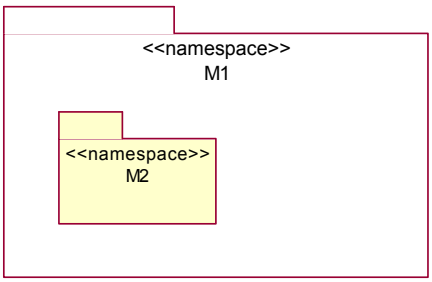


Abb. 46 UML-Notation für Namensraum

erwähnten Gründen entfernt. Eine vollständige Darstellung des virtuellen Metamodells der Konzepte Datentyp, Interfacetyp, Operation, Attribut und Ausnahme erfolgte in [OMG CORBA P].

Das Konzept Interfacetyp wird entsprechend des UML-Profils für CORBA-IDL mit der *Stereotype*-Definition ***CORBAInterface*** notiert. In diesem Fall ist das Entfernen des Präfix ***CORBA*** jedoch nicht erlaubt, da eine *Stereotype*-Definition ***Interface*** bereits in UML selbst enthalten ist. Zur Vermeidung dieses Konflikts wird hier stattdessen die *Stereotype*-Definition ***DOTInterface*** verwendet, wobei der Präfix ***DOT*** technologieunabhängig für *Distributed Object Technologie* steht.

Es werden im folgenden einige Beispiele für die Notation von ausgewählten Modellelementen, die Instanzen dieser Konzepte sind, präsentiert.

(Beispiel 7) Wenn ***S*** eine Strukturdefinition im Namensraum ***M1*** mit zwei Elementen ***m1*** und ***m2*** ist, wobei ***m1*** vom Typ ***string*** und ***m2*** vom Typ ***long*** sind, wird diese Situation wie in Abb. 47 dargestellt.

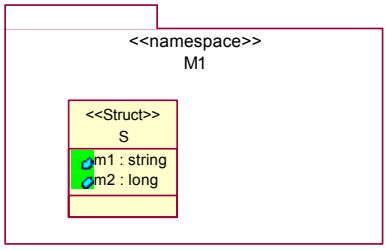


Abb. 47 UML-Notation für *Struct*-Datentyp

(Beispiel 8) Wenn ***E*** eine Ausnahme im Namensraum ***M1*** mit einem Element ***el1*** ist, wobei ***el1*** vom Typ ***S*** ist, wird diese Situation in UML wie in Abb.48 dargestellt:

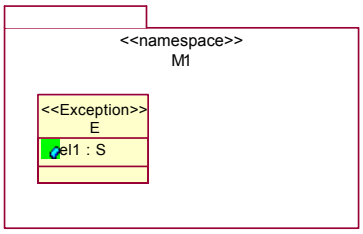


Abb. 48 UML-Notation für Ausnahme

(Beispiel 9) Sei ***i*** ein Interfacetyp im Namensraum ***M1*** mit einer Operation ***op1***, die einen Parameter ***p*** vom Typ ***long*** besitzt, der die Richtung ***in*** hat. Die Operation habe außerdem den Rückgabetyp ***S*** aus *Beispiel 8*. Diese Situation wird entsprechend Abb.49 notiert.

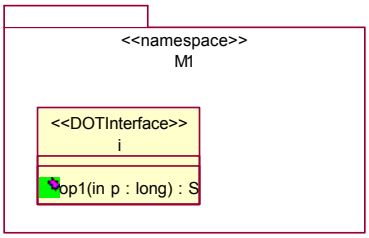


Abb. 49 UML-Notation für operationalen Interfacetyp

5.2.5.3 Signaltyp und Signalparameter

Signaltyp ist im Metamodell definiert als eine Metaklasse ***SignalDef***. Im UML-Profil wird für Signaltypen daher eine *Stereotype*-Definition vorgenommen, wobei das Basiselement das UML-Konzept ***class*** ist. Eigenschaften von Signaltypen, im Metamodell als Liste von *Property*-Definitionen erfaßt, werden als *Tagged-Value*-Definitionen in einem UML-Modell notiert. Da die genaue Angabe der Art der in Entwurfsmodellen zu erfassenden Signaltypeigenschaften aus Gründen der Technologieunabhängigkeit bewußt nicht erfolgt, werden die *Tagged-Value*-Definitionen nicht im virtuellen Metamodell des UML-Profils definiert, sondern erfolgen direkt in einem auf dem UML-Profil basierenden UML-Modell. Hierbei wird ausgenutzt, daß *Tagged-Value*-Definitionen in UML-Modellen im Kontext aller Modellelemente erfolgen können, unabhängig davon, ob sie zuvor im UML-Metamodell oder im virtuellen Metamodell eines UML-Profils deklariert wurden. Im Falle von Signaltypdefinitionen wird zusätzlich festgelegt, daß etwaige *Tagged-Value*-Definitionen auf eine konkrete *Property*-Liste an der Instanz der Metaklasse ***SignalDef*** abgebildet werden.

Signalparameter verweisen in einem Entwurfsmodell auf Instanzen der Metaklasse ***ValueTypeDef***. Da es sich bei diesem Konzept um ein aus CORBA-IDL bekanntes Konzept handelt, wird das virtuelle Metamodell für das UML-Profil und damit die Notation bereits durch das UML-Profil für CORBA-IDL festgelegt. Der Zusammenhang zwischen Signaltypen und ihren Signalparametern ist im Metamodell durch eine Liste von Elementen des Typs ***CarryField*** in einem Attribut der Metaklasse ***SignalDef*** modelliert. Im virtuellen Metamodell des UML-Profils wird dieser Sachverhalt durch eine *Stereotype*-Definition ***carry*** erfaßt, die auf dem UML-Metamodellelement ***Association*** beruht. Auf diese Weise ist in UML-Modellen die Darstellung des Zusammenhangs zwischen Signaltypen und Signalparametern durch eine Assoziation zwischen zwei UML-Klassen möglich, wobei die eine UML-Klasse eine Instanz der *Stereotype*-Definition ***Signal*** und die andere eine Instanz der *Stereotype*-Definition ***Value*** ist, wobei die Assoziation selbst eine Instanz der *Stereotype*-Definition ***carry*** ist.

Das durch die Metaklasse ***CarryField*** geforderte Attribut ***identifier*** ist bereits als Attribut an dem UML-Metamodellelement ***Association*** definiert, muß also nicht noch eingeführt werden. Das virtuelle Metamodell für Signaltypen und Signalparameter ist in Abb. 50 dargestellt.

(Beispiel 10) Sei ***V*** eine *Value*-Typefinition im Entwurfsmodell im Namensraum ***M1***, das die in *Beispiel 7* definierte Struktur ***S*** als Element mit dem Namen ***m1*** enthält. Sei weiterhin ***Sig1*** eine Signaltypdefinition im Entwurfsmodell im Namensraum ***M1***, die den Signalparameter ***carry_v*** vom Typ ***V*** transportiert. Dann wird diese Situation in einem UML-Modell entsprechend Abb. 51 dargestellt.

5.2.5.4 Erweiterter Interfacetyp

Das Konzept von erweiterten Interfacetypen wurde in *CoRE$_{CEPT}$* eingeführt, um die Unterstützung aller Interaktionsarten im Kontext eines Interfacetyps zu ermöglichen. Im Metamodell ist dafür die Metaklasse

Abb. 50 Virtuelles Metamodell für Signaltyp und Signalparameter

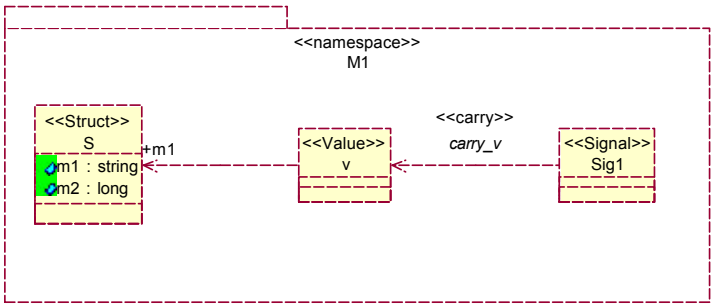


Abb. 51 UML-Notation für Signaltyp und Signalparameter

EnhancedInterfaceDef definiert, die von der im Metamodell von CORBA-IDL definierten Metaklasse ***InterfaceDef*** abgeleitet ist.

Natürlich soll das Konzept von erweiterten Interfacetypen auch durch die UML-Notation unterstützt werden. Dazu wird eine neue *Stereotype*-Definition vorgenommen, die von der *Stereotype*-Definition ***DOTInterface*** abgeleitet ist. Diese trägt den Namen ***EnhancedInterface***. Im virtuellen Metamodell des UML-Profils ist dieser Sachverhalt durch die Definition einer neuen Klasse ***EnhancedInterface*** mit einer entsprechenden Spezialisierungsrelation zu ***DOTInterface*** modelliert (vgl. Abb. 52).

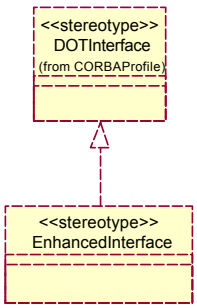


Abb. 52 Virtuelles Metamodell für Erweiterter Interfacetyp

In einem UML-Modell, das auf dem hier definierten UML-Profil basiert, wird eine Instanz von ***EnhancedInterfaceDef*** notiert durch eine Klasse mit dem Stereotyp ***EnhancedInterface***. Da die *Stereotype*-Definition ***DOTInterface*** auf dem UML-Metamodellelement ***class*** basiert, können Operations- und Attributdefinitionen im Kontext eines erweiterten Interfacetyps direkt in UML notiert werden. Eine Erweiterung oder

Anpassung durch Definitionen im virtuellen Metamodell des UML-Profils sind für die operationale Interaktionsart nicht erforderlich.

Die Notation der Zuordnung von Ausnahmen zu operationalen Interaktionselementen, die potentiell im Kontext einer entsprechenden Interaktion ausgelöst werden können, ist bereits durch das UML-Profil für CORBA-IDL definiert und basiert auf einer *Tagged-Value*-Definition für Operationen, die eine Liste der der Operation zuzuordnenden Ausnahmedefinitionen enthält. In dieser Arbeit wird diese Möglichkeit auf Attribute ausgedehnt, da das Konzept Attribut ebenso wie das Konzept Operation die Möglichkeit der Zuordnung von Ausnahmen vorsieht.

(Beispiel 11) Sei **I** eine erweiterte Interfacetypdefinition im Entwurfsmodell im Namensraum **M1**. Sei weiterhin eine Operation **op** ohne Rückgabetyp und Parameter an diesem Interfacetyp definiert, die die in *Beispiel 8* definierte Ausnahme **E** auslöst. Der Interfacetyp **I** enthält weiterhin ein *Readonly*-Attribut **attr** vom in *Beispiel 7* definierten strukturierten Datentyp **S**. Dann wird die Situation in UML wie in Abb.53 notiert:

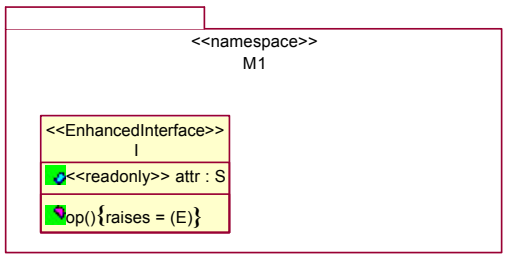


Abb. 53 UML-Notation für Erweiterter Interfacetyp

5.2.5.5 Interaktionselement, Consume- und Produce-Definition

Die Konzepte *Consume*-Definition und *Produce*-Definition dienen zur Modellierung von zu produzierenden oder zu konsumierenden Signalen eines Signaltyps. Im Metamodell von *CORE* sind diese Konzepte durch die Metaklassen **ConsumeDef** und **ProduceDef** modelliert, die jeweils auf eine Signaltypdefinition, also eine Instanz der Metaklasse **SignalDef** verweisen. Instanzen von **ConsumeDef** und **ProduceDef** dürfen im Kontext eines erweiteten Interfacetyps definiert werden.

Konzeptionell stellen die Konzepte *Consume*-Definition und *Produce*-Definition den Zusammenhang zwischen erweiteten Interfacetypen und Signaltypen unter einem identifizierbaren Namen her. Soll dieser Sachverhalt in UML notiert werden, so bietet sich das UML-Metamodellelement **Association** als Basis an, das geeignet spezialisiert werden muß. Das UML-Metamodellelement **Class** hätte an dieser Stelle auch benutzt werden können, der reale Sachverhalt, nämlich die Beziehung zwischen Interfacetyp und produziertem/ konsumierten Signaltyp wäre dann aber nicht mehr adäquat wiedergespiegelt.

Für die Konzepte *Consume*- und *Produce*-Definition wird jeweils eine *Stereotype*-Definition im UML-Profil vorgenommen. Im virtuellen Metamodell des UML-Profils werden diese *Stereotype*-Definitionen durch Klassen mit den Namen **Consume** und **Produce** modelliert, die auf dem UML-Metamodellelement **Association** basieren.

Assoziationen sind in UML-Modellen eindeutig identifizierbar, es muß keine Attributdefinition an den *Stereotype*-Definitionen zur Aufnahme des Bezeichners einer *Consume*- oder *Produce*-Definition bereitgestellt werden. Weitere Eigenschaften sind durch das Metamodell von *CORE* nicht vorgeschrieben und somit in UML auch nicht zu notieren.

Im Metamodell von *CORE* ist die abstrakte Metaklasse **InteractionElement** als gemeinsame Basis für alle im Kontext eines Interfacetyps spezifizierbaren Interaktionselemente definiert. Im UML-Profil wird diese Kon-

struktion nachvollzogen, wobei eine abstrakte *Stereotype*-Definition ***EnhancedInteractionelement*** erklärt wird (vgl. Abb. 54).

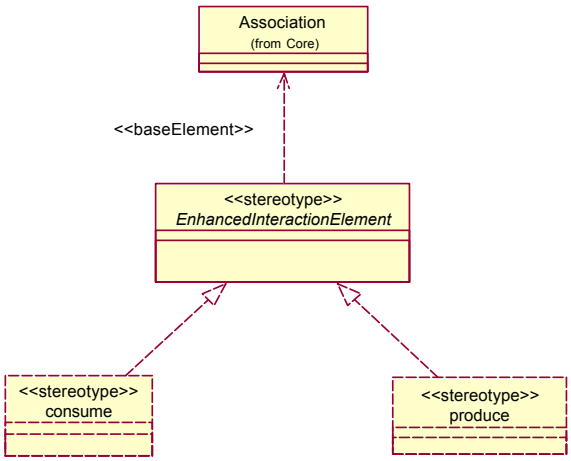


Abb. 54 Virtuelles Metamodell für *Consume*- und *Produce*-Definition

(Beispiel 12) Sei ***I*** der in *Beispiel 11* definierte Interfacetyp im Entwurfsmodell im Namensraum ***M1***. Sei weiterhin ***Sig1*** der in *Beispiel 7* definierte Signaltyp. Zwischen ***I*** und ***Sig1*** seien zwei *Produce*-Definitionen mit den Namen ***send_1_Sig*** und ***send_2_Sig*** sowie eine *Consume*-Definition mit dem Namen ***receive_Sig*** definiert. Diese Situation wird in UML wie in Abb. 55 notiert.

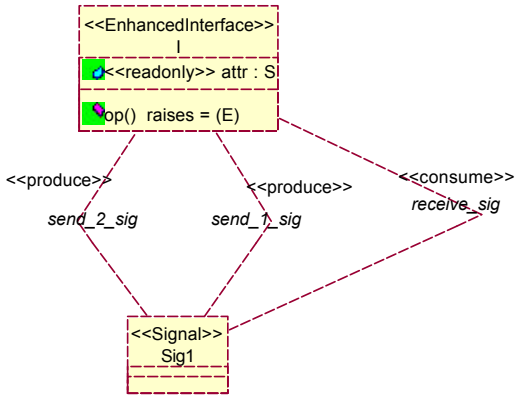


Abb. 55 UML-Notation für *Produce*- und *Consume*-Definition

5.2.5.6 Medientyp, Medium und Medienmenge

CORE$_{CEPT}$ beschränkt die möglichen Interaktionsarten an einem Interfacetyp nicht auf operationale und signalbasierte Interaktion. Über diese hinaus wurden zusätzliche Konzepte definiert, die die Deklaration von *Continous-Media*-Interaktionselementen an einem Interfacetyp gestatten. Diese Interaktionselemente basieren auf den Konzepten Medientyp, Medium und Medienmenge, deren Verwendung in einem UML-Modell zunächst erläutert wird, bevor die Notation der entsprechenden Interaktionselemente eingeführt werden kann.

Im Metamodell sind die Konzepte Medientyp, Medium und Medienmenge durch entsprechende Metaklassen definiert. Im UML-Profile werden daher *Stereotype*-Definitionen vorgenommen, die die Namen ***MediaType***, ***Medium*** und ***Mediaset*** tragen und auf dem UML-Metamodellelement ***Class*** basieren.

Die im Metamodell definierte Assoziation ***realized_by*** zwischen den Metaklassen ***MediumDef*** und ***MediaTypeDef*** kann dann in UML durch die Standard-*Stereotype*-Definition ***realize*** notiert werden, die wiederum auf dem UML-Metamodellelement ***Abstraction*** basiert.

Der Zusammenhang zwischen Medienmengen und den von ihnen aggregierten Medien ist im Metamodell durch eine Liste von Elementen des Typs ***MediumField*** in einem Attribut der Metaklasse ***SignalDef*** modelliert. In UML kann diese Aggregation durch Verwendung des UML-Metamodellelement ***Association*** notiert werden. Die konkrete Assoziation wird in einem UML-Modell zwischen zwei UML-Klassen definiert, wobei die eine UML-Klasse eine Instanz der *Stereotype*-Definition ***Mediaset*** und die andere eine Instanz der *Stereotype*-Definition ***Medium*** ist. Die durch das Metamodell festgelegte Möglichkeit der Identifikation von Instanzen von ***MediumField*** wird durch den Namen der Assoziation in UML notiert.

Eigenschaften von Medienmengen können analog zu Eigenschaften von Signaltypen in einem UML-Modell durch eine Menge von *Tagged-Value*-Definitionen notiert werden. Diese werden in einem UML-Modell derjenigen UML-Klasse zugeordnet, die die entsprechende Instanz der Metaklasse ***MediasetDef*** repräsentiert, der die Eigenschaften hinzugefügt werden sollen. Im virtuellen Metamodell des UML-Profils erscheinen für

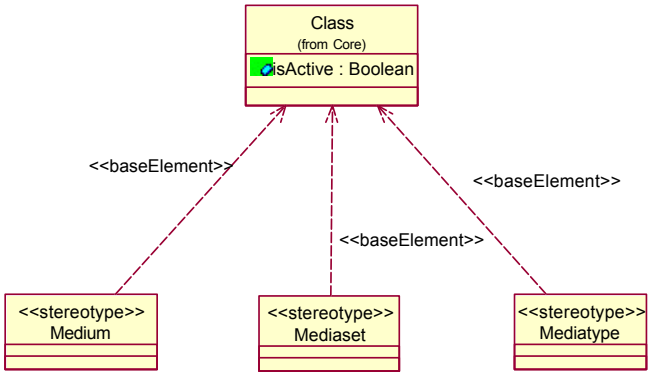


Abb. 56 Virtuelles Metamodell für Medium, Medienmenge und Medientyp

Eigenschaften von Medienmengen dagegen keine *Tagged-Value*-Definitionen, da die Art der Eigenschaften technologieabhängig ist und im Metamodell auch nicht festgelegt wurde. Das virtuelle Metamodell für Medium, Medientyp und Medienmenge ist in Abb. 56 dargestellt.

(Beispiel 13) Sei ***M*** eine Medienmengendefinition im Entwurfsmodell, die das Medium ***Audio*** unter dem Namen ***audio*** und das Medium ***Video*** untern dem Namen ***video*** aggregiert. Sei weiterhin ***Audio*** realisiert durch die Medientypdefinition ***MP3*** und ***Video*** durch die Medientypdefinitionen ***MPEG2*** und ***MPEG1***. Seien ferner alle diese Definitionen im Namensraum ***M1*** vorgenommen. Dann wird diese Situation in UML wie in Abb. 57 notiert.

5.2.5.7 Sink- und Source-Definition

Mit der oben eingeführten Notation für die Grundbestandteile der *Continous-Media*-Interaktion kann nun auch die Notation für die eigentlichen Interaktionselemente definiert werden. Diese Interaktionselemente sind in *CORE$_{CEPT}$* als Konzepte *Sink*- und *Source*-Definition eingeführt, die im Metamodell durch die Metaklassen ***SinkDef*** und ***SourceDef*** modelliert sind.

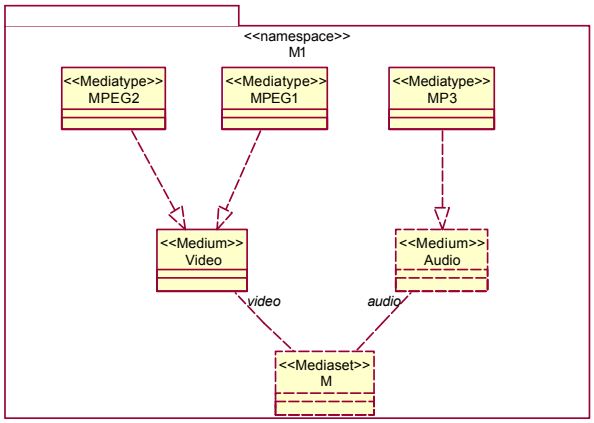


Abb. 57 UML-Notation für Medium, Medienmenge und Medientyp

Konzeptionell stellen Instanzen von ***SinkDef*** und ***SourceDef*** in einem Entwurfsmodell den Zusammenhang zwischen einer Instanz von ***MediasetDef*** und einer Instanz von ***EnhancedInterfaceDef*** her. Im UML-Profil von *CORE$_{TATIONS}$* werden zur Notation von *Sink*- und *Source*-Definitionen die *Stereotype*-Definitionen ***sink*** und ***source*** eingeführt, die auf dem UML-Metamodellelement ***Association*** basieren.

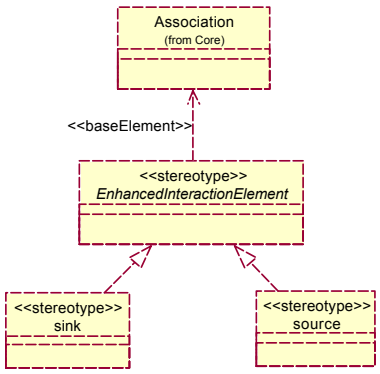


Abb. 58 Virtuelles Metamodell für *Sink*- und *Source*-Definition

Im virtuellen Metamodell werden die *Stereotype*-Definitionen ***sink*** und ***source*** als Klassen definiert. Die im Metamodell geforderte Möglichkeit der Identifikation ist bereits eine Eigenschaft des UML-Metamodellelements ***Association***, insofern werden im UML-Profil diesbezüglich keine *Tagged-Value*-Definitionen vorgenommen.

Da die Metaklassen ***SourceDef*** und ***SinkDef*** im Metamodell als Spezialisierungen der abstrakten Metaklasse ***Interactionelement*** definiert sind, wird auch im virtuellen Metamodell für das UML-Profile eine Spezialisierungsrelation von ***sink*** und ***source*** zu der abstrakten *Stereotype*-Definition ***EnhancedInteractionelement*** eingeführt. Das virtuelle Metamodell für *Sink* und *Source* ist in Abb. 58 dargestellt.

(Beispiel 14) Die in *Beispiel 12* dargestellte Situation sei insofern erweitert, als daß der Interfacetyp ***I*** eine *Source*-Definition ***av_out*** und eine *Sink*-Definition ***av_in*** zu der Medienmengendefinition ***M*** definiert, die in *Beispiel 13* definiert wurde. Diese Situation wird in UML wie in Abb. 59 notiert.

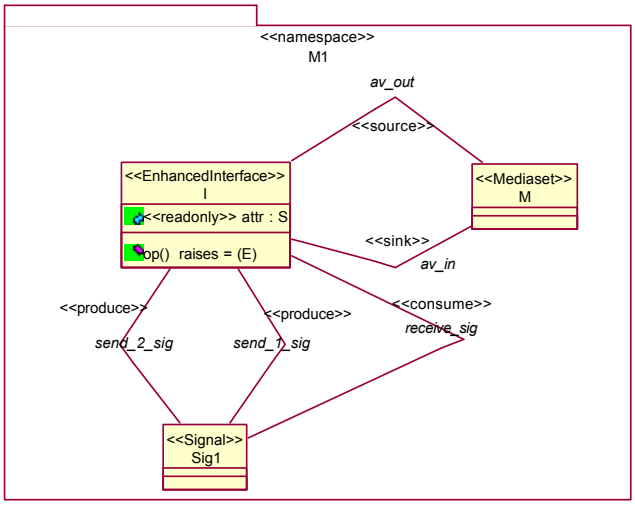


Abb. 59 UML-Notation für *Sink*- und *Source*-Definition

5.2.5.8 CO-Typ, Supports- und Requires-Relation

CO-Typen bilden neben den erweiterten Interfacetypen ein Basiskonzept von *CoRE*. Im Metamodell ist das Konzept CO-Typ als eine Metaklasse modelliert, die von der Metaklasse ***InterfaceDef*** abgeleitet ist. Dieser Ableitungsschritt gewährleistet, daß an CO-Typen operationale Interaktionselemente definiert werden können, die u.a. zur Konfiguration von Instanzen der CO-Typen verwendbar sind.

In Analogie zu dieser Spezialisierungsrelation im Metamodell wird zur Notation von CO-Typen eine *Stereotype*-Definition ***CO*** eingeführt, die von der *Stereotype*-Definition ***DOTInterface*** des UML-Profils für CORBA-IDL abgeleitet ist. Diese *Stereotype*-Definition basiert auf dem UML-Metamodellelement ***Class***. Damit können in einem UML-Modell an Instanzen der *Stereotype*-Definition ***CO*** Operationen notiert werden, die durch das Metamodell geforderte Semantik ist damit in UML-Modellen darstellbar.

Durch die indirekte Ableitung der *Stereotype*-Definition ***CO*** von dem UML-Metamodellelement ***class*** können in einem UML-Modell auch Assoziationen zwischen Instanzen von ***CO*** und anderen UML-Elementen, die auf ***class*** basieren, definiert werden. Diese Eigenschaft kann zunächst zur Darstellung des Zusammenhangs zwischen CO-Typen und Interfacetypen verwendet werden. Interfacetypen können von CO-Typen entweder unterstützt oder benötigt werden. Das ist im Metamodell durch die Assoziationen ***supports*** und ***requires*** zwischen den Metaklassen ***COTypeDef*** und ***InterfaceDef*** modelliert. Beide Assoziationen werden im UML-Profil durch *Stereotype*-Definitionen (***supports*** und ***requries***) unterstützt, die auf dem UML-Metamodellelement ***Association*** basieren.

Im virtuellen Metamodell des UML-Profils werden die *Stereotype*-Definitionen ***CO***, ***supports*** und ***requires*** als Klassen modelliert, wobei ***CO*** von der Klasse ***DOTInterface*** abgeleitet ist und ***supports*** und ***requires*** auf der Klasse ***Association*** des UML-Metamodells basieren. Das virtuelle Metamodell des UML-Profils für CO-Typen, *Supports*- und *Requires*-Relation ist in Abb. 60 dargestellt.

> *(Beispiel 15)* Sei ***CO1*** eine CO-Typdefinition im Entwurfsmodell im Namensraum ***M1*** und sei eine Operation ***config*** mit einem Parameter ***p*** des Typs ***string*** ohne Rückgabetyp mit der Richtung ***in*** an ***CO1*** definiert. Seien zusätzlich eine Assoziation ***supports*** und eine Assoziation ***requires*** zu der Interfacetypdefinition ***I*** aus *Beispiel 11* definiert. Diese Situation wird in UML wie in Abb. 61 notiert.

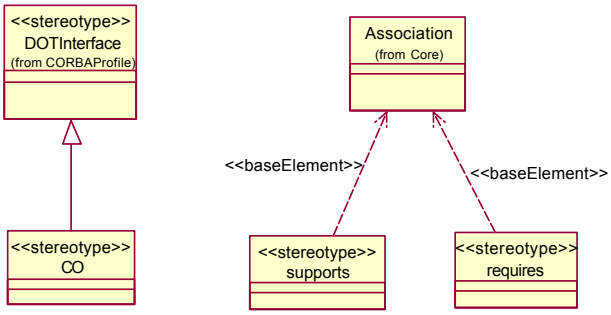


Abb. 60 Virtuelles-Metamodell für CO-Typ, *Supports*- und *Requires*-Relation

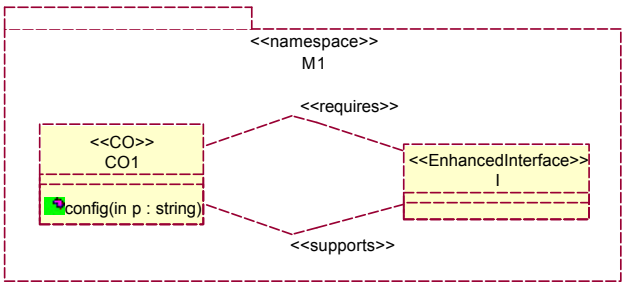


Abb. 61 UML-Notation für CO-Typ, *Supports*- und *Requires*-Relation

5.2.6 Konfigurationssicht

Die Konzepte der Konfigurationssicht sind - ebenso wie die bisher dargestellten Konzepte - struktureller Art und damit typbasiert. Zur Darstellung wird daher die UML-Klassennotation herangezogen, wobei geeignete *Stereotype*-Definitionen die Abbildung von UML-Modellen auf Entwurfsmodelle basierend auf dem Metamodell von *CORE* gewährleisten.

5.2.6.1 Port-, Provided- und Used-Port-Definition

Das Konzept *Port*-Definition wurde in *CORE$_{CEPT}$* integriert, um zur Ausführungszeit eines verteilten Softwaresystems das Hinterlegen bzw. Beschaffen von Interfacereferenzen auf von CO-Typen unterstützte bzw. benötigte Interfacetypen zu ermöglichen. Konzeptionell stellt eine *Port*-Definition eine Relation zwischen Instanzen der Konzepte CO-Typ und Interfacetyp in einem Entwurfsmodell her. Im Metamodell ist das Konzept *Port*-Definition durch eine abstrakte Metaklasse ***PortDef*** definiert, die im Kontext eines konkreten CO-Typs eindeutig identifizierbar ist. Zur Notation von *Port*-Definitionen bietet sich eine Lösung an, die auf der Verwendung des UML-Metamodellelements ***Association*** beruht. Für das Konzept *Port*-Definition wird eine *Stereotype*-Definition im UML-Profil von *CORE$_{TATIONS}$* vorgenommen, die den Namen ***port*** trägt und auf dem UML-Metamodellelement ***Association*** basiert. Die Möglichkeit der Identifikation von *Port*-Definitionen wird über den Namen der ***Association***-Definition ermöglicht.

Im virtuellen Metamodell ist die *Stereotype*-Definition ***port*** als abstrakte Klasse ***port*** mit einer Abhängigkeit der Art ***baseElement*** zu der Klasse ***Association*** aus dem UML-Metamodell definiert. Die abstrakte Definition erfolgte, weil auch ***PortDef*** im Metamodell von *CORE* abstrakt definiert ist, es also keine Instanzen in Entwurfsmodellen geben kann.

Da die *Stereotype*-Definition ***port*** bereits auf dem UML-Metamodellelement ***Association*** basiert, brauchen keine weiteren Elemente im UML-Profil eingeführt werden, um die im Metamodell definierte ***Association***

zwischen ***PortDef*** und ***InterfaceDef*** darzustellen. Zusätzlich muß noch eine Möglichkeit zur Notation der Vielfachheit von *Port*-Definitionen geschaffen werden, die dann zu einer Belegung des Attributes ***multiple*** an einer Instanz von ***Portdef*** in einem Entwurfsmodell führt. Die Notation dieses Attributes kann durch eine *Tagged-Value*-Definition an der *Stereotype*-Definition ***port*** erfolgen - eine allerdings nicht elegante Art der Notation in UML. Vielmehr ist es in UML vorgesehen, die Vielfachheit einer Assoziation durch die Belegung eines entsprechenden Attributes am jeweiligen Assoziationsende auszudrücken und dies in einem UML-Modell durch eine Annotation an dem die Assoziation definierenden Symbol (i.allg. ein Pfeil) zu notieren. Diese Möglichkeit wird hier ausgenutzt, da die *Stereotype*-Definition ***port*** auf dem UML-Metamodellelement ***Association*** basiert. Es wird festgelegt, daß, falls das Attribut ***multiple*** einer Instanz von ***PortDef*** in einem Entwurfsmodell den Wert ***true*** hat, dies in UML durch die Vielfachheit ***** an demjenigen Ende der Assoziation ***port*** notiert wird, deren UML-Klasse eine Instanz der *Stereotype*-Definition ***DOTInterface*** ist. Ansonsten ist die Vielfachheit ***1*** anzugeben.

Das Metamodell von *CORE* definiert zwei weitere Metaklassen als Spezialiserungen von ***PortDef***: ***UsePortDef*** und ***ProvidePortDef***. Konzeptionell bedeutet eine Instanz von ***UsePortDef*** in einem Entwurfsmodell, das zur Ausführungszeit auf der Grundlage dieser *Port*-Definition eine Interfacereferenz hinterlegt werden kann und ***ProvidePortDef***, daß eine Interfacereferenz verfügbar gemacht wird. Beide Konzepte werden in das UML-Profil als *Stereotype*-Definitionen ***use*** und ***provide*** integriert, die die *Stereotype*-Definition ***port*** spezialisieren. Im vir-

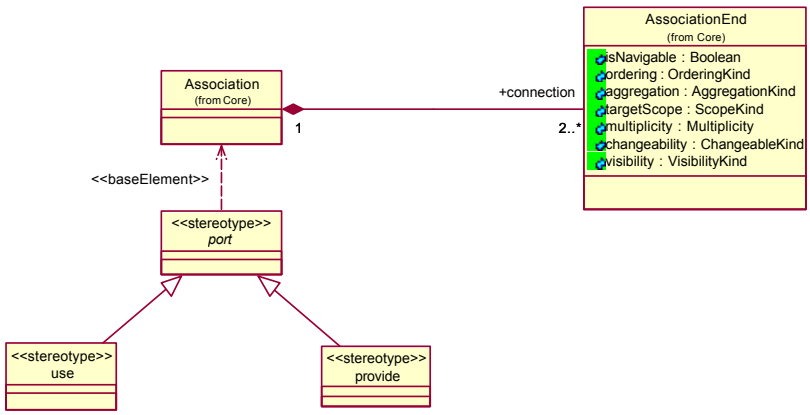


Abb. 62 Virtuelles Metamodell für *Port*-Definition

tuellen Metamodell des UML-Profils ist dieser Sachverhalt durch eine Klasse ***use*** und eine Klasse ***provide*** modelliert, die von der Klasse ***port*** abgeleitet sind.

Das virtuelle Metamodell für *Port*-Definitionen ist in Abb. 62 dargestellt.

> *(Beispiel 16)* Sei ***CO1*** der CO-Typ im Entwurfsmodell, der in *Beispiel 15* mit einer *Supports*-Relation zu dem Interface ***I*** definiert wurde. Auf der Basis der *Supports*-Relation sei eine *Provide-Port*-Definition mit dem Namen ***serve*** definiert, sie habe die Vielfachheit "***1***". Diese Situation wird in UML wie in Abb. 63 notiert.

Wie in Abb. 63 ersichtlich, führt die Notation einer ***supports***- *und* einer ***provide***-Assoziation sowie die Notation einer ***requires***- *und* einer ***use***-Assoziation zu einem hohen Aufwand bei der Erstellung der UML-Diagramme. Es wird daher erlaubt, auf die Angabe der supports-Assoziation zu verzichten, wenn eine ***provide***-Assoziation zwischen dem beteiligten CO-Typ und dem Interfacetyp vorgenommen wird. Ebenfalls kann auf ***requires*** verzichtet werden, wenn eine entsprechende ***use***-Assoziation notiert wird. Im Konzeptgraph für das Entwurfsmodell sind aber sowohl Instanzen der Assoziationen ***supports*** und ***requires*** als auch Instanzen der jeweiligen ***UsePortDef*** und ***ProvidePortDef*** zu erzeugen.

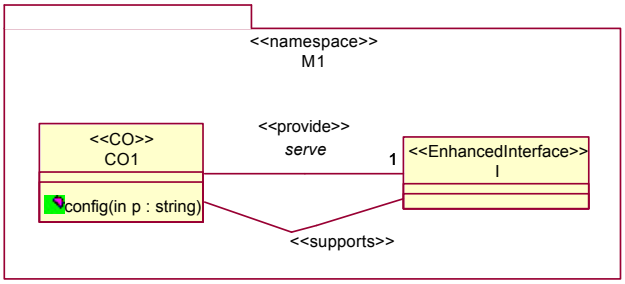


Abb. 63 UML-Notation für *Port*-Definition

5.2.7 Implementierungssicht

Die Konzepte der Implementierungssicht stellen in der Mehrzahl strukturelle Konzepte dar. Zu ihrer Darstellung bietet sich wiederum die Verwendung der UML-Klassennotation an. Verhaltensaspekte, die ebenfalls in der Implementierungssicht enthalten sind, können als Annotationen in Form von *Tagged-Value*-Definitionen zu der Klassennotation hinzugefügt werden.

5.2.7.1 Artefakt

Das Verhalten eines COs wird beim Ausführen einer Softwarekomponente durch Instanziierung programmiersprachlicher Konstrukte erbracht, die im Entwurfsmodell durch Artefakte abstrahiert sind. Das Konzept Artefakt ist im Metamodell von *CORE* durch eine Metaklasse ***AtrifactDef*** repräsentiert, wobei die Möglichkeit der Identifikation für Artefaktdefinitionen im Rahmen von Entwurfsmodellen gefordert wird. Im UML-Profil von *CORE* wird das Konzept Artefakt durch eine *Stereotype*-Definition ***Artifact*** realisiert, die auf dem Standard UML-Metamodellelement ***class*** basiert. Damit ist die Möglichkeit der Identifikation von Artefakten über den Namen der UML-Klasse ermöglicht. Generalisierungsrelationen zwischen Artefakten (im Metamodell erfaßt durch die Assoziation ***artifactDerivedFrom***) werden durch die UML-Metamodellrelation ***Generalization*** notiert, die zwischen UML-Klassen definiert ist.

Im virtuellen Metamodell des UML-Profils wird ***Artifact*** als Klasse definiert, die eine Abhängigkeit der Art ***baseElement*** zu der Klasse ***class*** des UML-Metamodells besitzt. Dies ist in Abb.64 verdeutlicht.

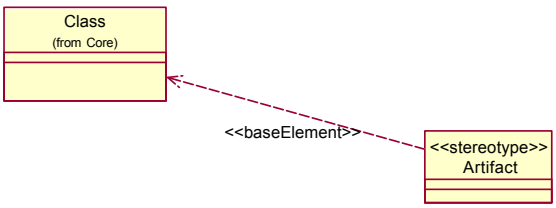


Abb. 64 Virtuelles Metamodell für Artefakt

5.2.7.2 Implements-Relation

Die Relation zwischen CO-Typen und den Artefakten, die das Verhalten an den von ihnen unterstützten bzw. benötigten Interfacetypen realisieren, ist im Metamodell durch die Einführung einer Assoziation ***implements*** zwischen ***ArtifactDef*** und ***COTypeDef*** berücksichtigt. Im Rahmen des UML-Profils wird für die Assoziation ***implements*** eine *Stereotype*-Definition ***implements*** vorgenommen, die auf dem UML-Metamodellelement ***Association*** basiert. Da die beteiligten Elemente dieser Relation (CO-Typen und Artefakte) im UML-Profil als Klassen notiert werden, ist diese Definition von ***implements*** adäquat.

Im virtuellen Metamodell des UML-Profils von *CoRE* wird die *Stereotype*-Definition ***implements*** als Klasse definiert, die eine Abhängigkeit der Art ***baseElement*** zu der Klasse ***Association*** aus dem UML-Metamodell besitzt (vgl. Abb. 65).

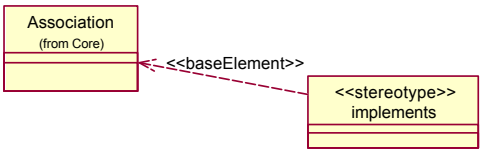


Abb. 65 Virtuelles Metamodell für *implements*

In UML-Modellen ist bei der Verwendung der *Stereotype*-Definition ***implements*** das Attribut ***aggregation*** desjenigen Endes der Assoziation, das dem beteiligten CO-Typ (*Stereotype*-Definition CO) zugeordnet ist, mit dem Wert ***aggregate*** zu belegen. Damit kann dargestellt werden, daß das Verhalten von CO-Typen durch eine *Menge* von Artefakten erbracht wird, die in diesem Sinne Bestandteile des CO-Typs sind. Artefakte dürfen im Kontext verschiedener CO-Typen verwendet werden, deshalb kann nicht ***composite*** als Wert des Attributes ***aggregation*** verwendet werden.

(Beispiel 17) Sei der CO-Typ ***CO1*** implementiert durch die Artefakte ***A1***, ***A2*** und ***A3***. Seinen weiterhin alle diese Artefakte im Namensraum ***M1*** definiert. Dies wird in UML wie in Abb. 66 dargestellt.

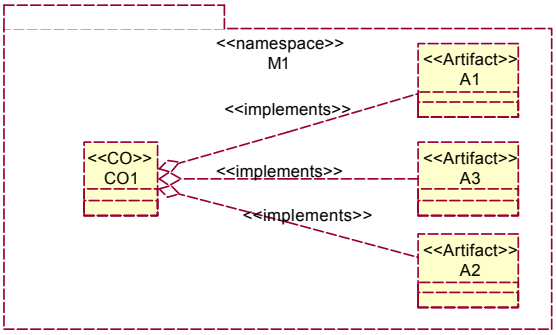


Abb. 66 UML-Notation für Artefakt und *implements*

5.2.7.3 Implementierungselement

Mit der Definition der Notation von Artefakten und der *Implements*-Relation ist nunmehr die Voraussetzung geschaffen, um auch die Definition der Notation für Implementierungselemente im Rahmen des UML-Profils von *CoRE* vorzunehmen. Im Metamodell ist das Konzept Implementierungselement durch eine Metaklasse ***ImplementationElementDef*** modelliert, für deren Instanzen die eindeutige Möglichkeit der Identifikation im Rahmen des definierenden Artefakts (Instanz von ***ArtifactDef***) gefordert wird. Weiterhin ist festgelegt, daß durch ein Attribut der Metaklasse ***ImplmentationElementDef*** eine Fallunterscheidung über die Art des Implementierungselements (***use*** oder ***supply***) erfolgt.

Für das Konzept Implementierungselement wird im UML-Profil eine *Stereotype*-Definition ***implelem*** vorgenommen, die auf dem Standard UML-Metamodellelement ***Operation*** basiert. Der Grund hierfür besteht darin, daß Implementierungselemente das Verhalten für ein ihnen zugeordnetes Interaktionselement erbringen, ihrem Wesen nach also keine Klassifizierungselemente, sondern Verhaltenselemente darstellen. Da das Konzept Artefakt in UML durch eine Klasse notiert wird, ist die Zuordnung von Implementierungselementen (notiert durch Operationen) zu Artefakten (notiert durch Klassen) bereits durch die UML-Semantik erreicht, es brauchen keine diesbezüglichen Erweiterungen im UML-Profil vorgenommen werden.

Eine Besonderheit ist jedoch, daß eventuell in UML notierte Parameter, Ausnahmen oder Rückgabetypen für als Operationen notierte Implementierungselemente nicht in ein Entwurfsmodell, das auf dem Metamodell von *CORE* basiert, übernommen werden. Der Grund besteht darin, daß die Signatur der Operation von der Art des zugeordneten Interaktionselements einerseits und von der verwendeten Realisierungstechnologie, insbesondere der *Component-Support*-Plattform abhängig ist. Es bleibt somit den Regeln für die automatische Ableitung von Softwarekomponenten aus Entwurfsmodellen überlassen, eine geeignete Signatur zu definieren.

Im virtuellen Metamodell des UML-Profils wird die *Stereotype*-Definition ***implelem*** durch eine Klasse ***implelem*** modelliert, die eine Abhängigkeit der Art ***baseElement*** zu der Klasse ***Operation*** des UML-Metamodells besitzt.

Die Festlegung des Falles (***use*** oder ***supply***) den ein Implementierungselement in Bezug auf eine Interaktionselement implementieren soll, wird im UML-Profil durch eine *Tagged-Value*-Definition ***case*** berücksichtigt. Der Wert dieser *Tagged-Value*-Definition muß entweder ***use*** oder ***supply*** sein.

Im virtuellen Metamodell wird die *Tagged-Value*-Definition ***case*** als Attribut an der Klasse ***implelem*** definiert, wobei der Wert des Attributes eine *Enumeration*-Definition mit den Enumeratoren ***use*** und ***supply*** ist.

Im Metamodell von *CORE* ist der Bezug zwischen Implementierungselementen und Interaktionselementen durch eine Assoziation modelliert, wobei von einer Instanz eines Implementierungselements in einem Entwurfsmodell zu der zugeordneten Instanz des Interaktionselements navigiert werden kann. Entsprechend den in Abschnitt 5.2.4.1 aufgestellten Regeln, ist diese Assoziation im UML-Profil durch eine *Stereotype*-Definition darzustellen, die auf dem UML-Metamodellelement ***Association*** basiert. Die Schwierigkeit ist allerdings, daß die Interaktionselemente Operation und Attribut durch die UML-Metamodellelmente ***Operation*** und ***Attribute*** dargestellt werden, die wiederum nicht von dem UML-Metamodellelement ***Classifier*** abgeleitet sind und somit nicht an Assoziationen teilhaben dürfen. Im Rahmen von *CORE* muß für die Darstellung der Relation zwischen Interaktionselementen und Implementierungselementen eine andere Lösung gefunden werden. Das gelingt durch die Delegierung der Relation zwischen diesen Konzepten auf die jeweiligen definierenden Kontexte: Interfacetyp und Artefakt. Konzeptionell realisiert ein Artefakt Teile eines Interfacetyps, wobei im Entwurfsmodell durch die Definition der Assoziation zwischen Implementierungselement und Interaktionselement gerade die Zuordnung für diese Realisierung festgelegt wird. Im UML-Profil wird zur Darstellung dieses Zusammenhangs die Standard-*Stereotype*-Definition ***realize*** als Spezialisierung des UML-Metamodellelements ***Abstraction*** benutzt. In dem Attribut ***mapping*** dieses UML-Metamodellelements kann die Zuordnung zwischen Interaktionselementen des Interfacetyps und den Implementierungselementen des Artefakts notiert werden.

Das virtuelle Metamodell des UML-Profils für Implementierungselemente ist in Abb. 67 dargestellt.

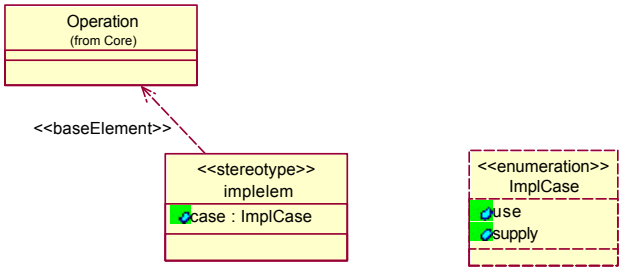


Abb. 67 Virtuelles Metamodell für Implementierungselement

(Beispiel 18) Die in *Beispiel 17* dargestellte Situation sei wie folgt verändert, die Notation ist in Abb. 68 in UML notiert:

- ***CO1*** ist durch Artefakt ***A1*** und ***A2*** implementiert.

- Im Kontext von ***A1*** seien die Implementierungselemente ***attr_impl***, ***op_impl***, ***send_1_sig_impl***, ***send_2_sig_impl***, ***receive_sig_impl***, ***av_out_impl*** und ***av_in_impl*** für die Interaktionselemente ***attr***, ***op***, ***send_1_sig***, ***send_2_sig***, ***receive_sig_impl***, ***av_out*** und ***av_in*** definiert. Für diese Implementierungselemente sei der Implementierungsfall ***supply***.
- Im Kontext von ***A2*** seien die Implementierungselemente ***receive_1_sig_impl***, ***receive_2_sig_impl***, ***send_sig_impl***, ***av_in_impl*** und ***av_out_impl*** für die Interaktionselemente ***send_1_sig***, ***send_2_sig***, ***av_out*** und ***av_in*** definiert. Für diese Implementierungselemente sei der Implementierungsfall ***use***.

Die oben angegebene Zuordnung zwischen Interaktionselementen und Implementierungselementen wird als Bestandteil der *Realize*-Relation zwischen der Artefaktklasse (UML-Klasse zur Darstellung des Artefakts) und der Interfacetypklasse (UML-Klasse zur Darstellung des Interfacestyps) im UML-Modell definiert. Die Möglichkeit einer solchen Zuordnung ist in UML durch das Attribut ***mapping*** an der Metaklasse ***Abstraction*** des UML-Metamodells gegeben.

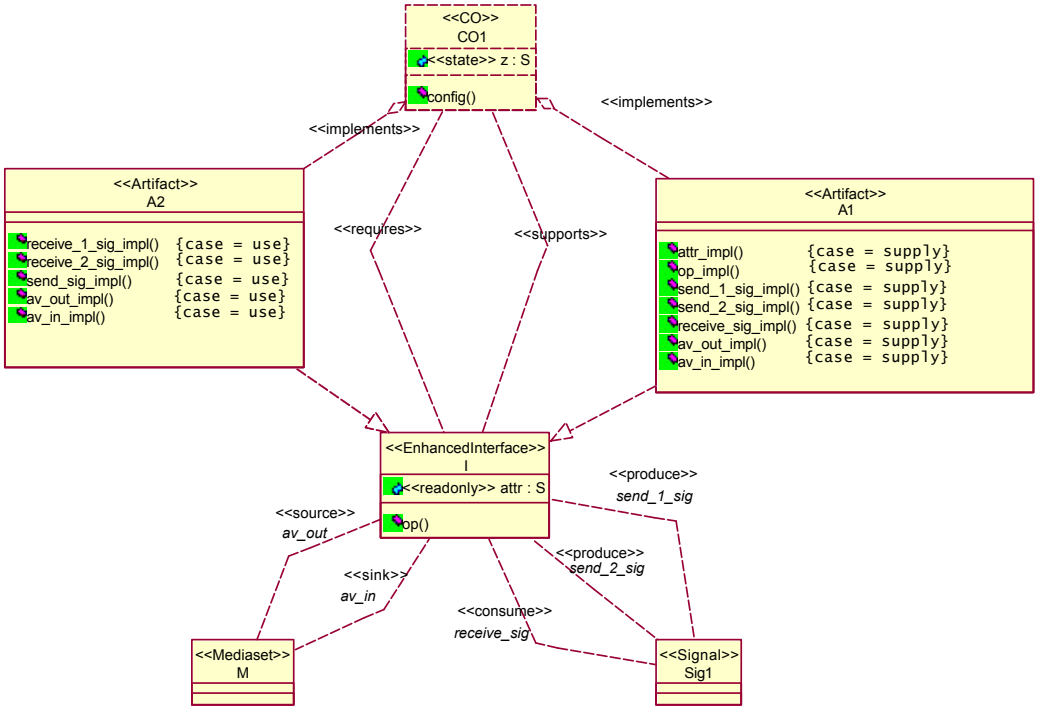


Abb. 68 Beispiel für Implementierungselement

5.2.7.4 Zustandsattribut

Die *Business-Logic*-Implementierung der Implementierungselemente einer Artefaktdefinition kann den Zugriff auf Zustandsinformation der zu realisierenden CO-Typdefinition erforderlich machen. Im Metamodell erfolgt die Beschreibung der Zustandsinformation typbasiert durch die Definition von Zustandsattributen im Kontext von CO-Typen, wobei Instanzen der Zustandsattribute zur Ausführungszeit gerade eine konkrete Zustandsinformation repräsentieren. Das Konzept Zustandsattribut ist im Metamodell von *CoRE* durch eine Metaklasse ***StateDef*** repräsentiert, die von der Metaklasse ***Contained*** abgeleitet ist. Definierender Container für ***StateDef***-Instanzen in Entwurfsmodellen ist immer eine Instanz von ***COTypeDef***.

Für ***StateDef*** wird im UML-Profil eine neue *Stereotype*-Definition ***state*** eingeführt. Zur Notation kann das UML-Metamodellelement ***Attribut*** genutzt werden, da Attribute in UML-Modellen an Klassen definiert sind und CO-Typen gerade durch UML-Klassen notiert werden. Weiterhin besitzen UML-Attribute Typen und

sind im Kontext ihrer definierenden UML-Klasse eindeutig identifizierbar. Aus diesem Grund basiert die *Stereotype*-Definition ***state*** auf dem UML-Metamodellelement ***Attribute***.

Im virtuellen Metamodell des UML-Profils wird eine Klasse ***state*** definiert, die auf der Klasse ***Attribute*** des UML-Metamodells basiert. Das virtuelle Metamodell des UML-Profils für Zustandsattribute ist in Abb. 69 dargestellt.

Im Metamodell ist für die Zuordnung derjenigen Zustandsattribute, deren Instanzen für die Realisierung eines Implementierungselements im Kontext einer Artefaktdefinition benötigt werden, eine Assoziation zwischen den Metaklassen ***ImplementationelementDef*** und ***StateDef*** definiert. Da Implementierungselemente in UML durch Operationen und Zustandsinformationen durch Attribute notiert werden, kann diese Assoziation nicht durch eine entsprechende Assoziation in UML dargestellt werden. Statt dessen werden die einem Implementierungselement zugeordneten Zustandsattribute durch eine *Tagged-Value*-Definition ***state*** notiert. Wert dieser *Tagged-Value*-Definition in einem UML-Modell ist eine Liste mit den Bezeichnern derjenigen UML-Attribute, mit denen die Zustandsattribute dargestellt wurden.

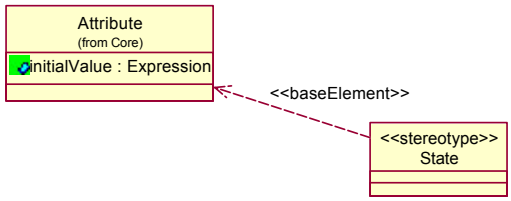


Abb. 69 Virtuelles Metamodell für Zustandsattribut

Im virtuellen Metamodell des UML-Profils wird die zusätzliche *Tagged-Value*-Definition durch ein Attribut ***state*** an der Klasse ***implelem*** definiert. Der Typ dieses Attributes ist ***string***, Wert in einem UML-Modell ist die Liste aller dem notierten Implementierungselement zugeordneten Zustandsattribute. Die Darstellung in Abb.67 wird um eine entsprechende Attributdefinition erweitert.

(Beispiel 19) Sei *Beispiel 18* derart erweitert, daß ein Zustandsattribut ***z*** des Typs ***S*** an ***CO1*** existiert, sowie daß ***z*** im Kontext der Implementierungselemente ***attr_impl*** und ***op_impl*** benötigt wird. Diese Situation ist in Abb. 70 dargestellt.

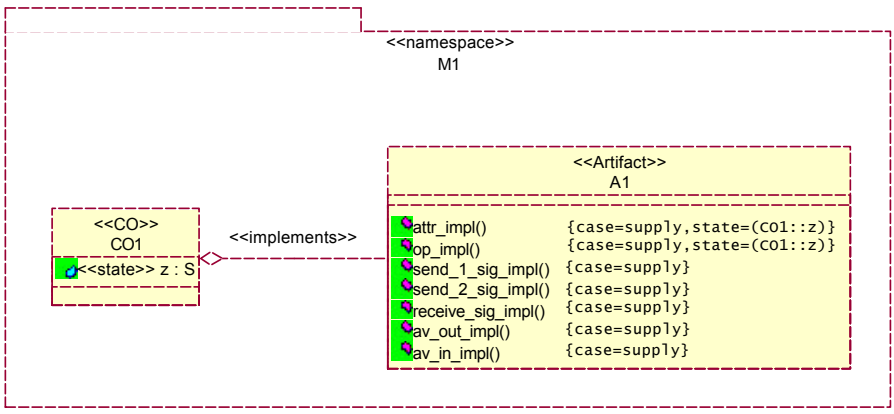


Abb. 70 Notation für Zustandsattribut

5.2.7.5 Instanziierungsmuster

Das Konzept Instanziierungsmuster dient zur Auswahl eines Verfahrens, die zur Instanziierung von Artefakten zur Ausführungszeit eingesetzt werden soll. Die Menge der zur Verfügung stehenden Werte wird im Metamodell durch einen Aufzählungstyp (*Enumeration*) beschrieben. Weiterhin ist im Metamodell ein Attribut an der Metaklasse ***ArtifactDef*** definiert, dessen Werte an Instanzen von ***ArtifactDef*** in Entwurfsmodellen gerade die für das Artefakt anzuwendende Instanziierungsstrategie bestimmen.

Im UML-Profil wird eine *Tagged-Value*-Definition für die Notation von Instanziierungsmustern von Artefakten eingeführt. Das virtuelle Metamodell des UML-Profils wird dementsprechend um ein Attribut ***instantiation_policy*** an der Klasse ***Artifact*** erweitert. Der Typ dieses Attributes ist eine *Enumeration*-Definition mit den im Metamodell bereits definierten Enumeratoren ***ARTIFACT_POOL***, ***ARTIFACT_PER_REQUEST***, ***SINGLETON*** und ***USER_DEFINED***. Das virtuelle Metamodell für Instanziierungsmuster ist in Abb. 71 dargestellt.

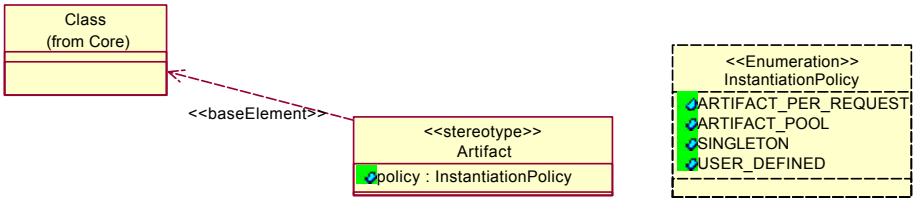


Abb. 71 Virtuelles Metamodell für Instanziierungsmuster

(Beispiel 20) Für die Artefaktdefinition ***A1*** aus *Beispiel 19* sei das Instanziierungsmuster ***SINGLETON*** angegeben. Diese Situation ist in Abb. 72 notiert.

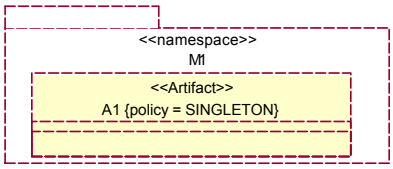


Abb. 72 Notation für Instanziierungsmuster

5.2.8 Deployment-Sicht

5.2.8.1 Softwarekomponente und Realize-Relation

Eine Softwarekomponente enthält Codemodule, deren Ausführung die Identität, den Zustand und das Verhalten von COs realisieren. Im Metamodell ist zur Repräsentation von Softwarekomponenten die Metaklasse ***ComponentDef*** eingeführt, die eine Assoziation zur Metaklasse ***COTypeDef*** besitzt. Durch Instanziierung dieser Assoziation kann erfaßt werden, welche CO-Typen von einer Softwarekomponente realisiert werden.

Das Konzept Softwarekomponente ist in UML bereits als UML-Metamodellelement ***Component*** berücksichtigt. Dieses Konzept wird zur Notation von Softwarekomponenten im Rahmen des UML-Profils genutzt. Es wird eine *Stereotype*-Definition ***SoftwareComponent*** eingeführt, die auf dem UML-Metamodellelement ***Component*** basiert. Im virtuellen Metamodell ist diese *Stereotype*-Definition reflektiert durch eine entsprechende Klasse, die eine Abhängigkeit der Art ***<<baseElement>>*** zu der Klasse ***Component*** des UML-Metamodells besitzt.

Die Assoziation zwischen den Klassen ***ComponentDef*** und ***COTypedef*** im Metamodell wird im UML-Profil durch die Standard-UML-Assoziation ***ElementResidence*** zwischen den UML-Metaklassen ***Component*** und ***Modelelement*** realisiert. Es braucht also kein zusätzliches Konstrukt in das UML-Profil aufgenommen zu werden, da COs durch die *Stereotype*-Definition ***COType*** notiert sind, die indirekt auf der UML-Metaklasse ***Modelelement*** basiert und somit die Assoziation ***ElementResidence*** zur Notation eingesetzt werden kann.

Das virtuelle Metamodell für ***Component*** ist in Abb.73 dargestellt.

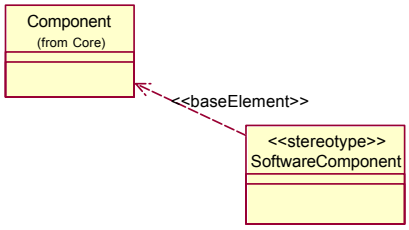


Abb. 73 Virtuelles Metamodell für Softwarekomponente

(Beispiel 21) Sei der CO-Typ ***CO1*** wie in *Beispiel 19* definiert. Sei weiterhin eine Softwarekomponente ***CO1Impl*** definiert, die ***CO1*** realisiert. Dann wird diese Situation in UML wie in Abb. 74 notiert.

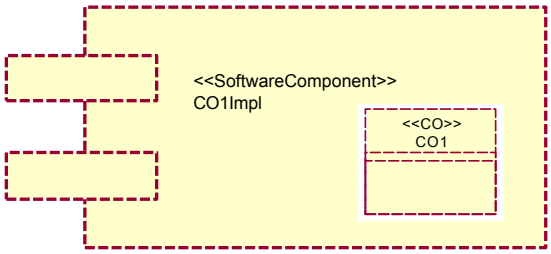


Abb. 74 Notation für Softwarekomponente

Softwarekomponenten können Abhängigkeiten von anderen Softwarekomponenten haben und Eigenschaften besitzen. Zur Modellierung der Abhängigkeiten ist im Metamodell von *CoRE* die Metaklasse ***ComponentDependencyDef*** definiert, zur Modellierung von Eigenschaften wurde das Attribut ***properties*** (*Property*-Liste) an der Metaklasse ***ComponentDef*** eingeführt.

Zur Notation von Abhängigkeiten zwischen Softwarekomponenten kann das UML-Metamodellelement ***Dependency*** benutzt werden. Die Verwendung von ***Dependency***-Relationen zwischen ***Component***-Elementen in einem UML-Modell ist durch die UML-Semantik bereits erklärt. Da die Metaklasse ***ComponentDependencyDef*** keine weiteren Eigenschaften besitzt, ist die Bereitstellung der Notation durch das UML-***Dependency***-Konstrukt ausreichend. Erweiterungen des virtuellen Metamodells des UML-Profils sind hierfür nicht vorzunehmen.

Eigenschaften von Softwarekomponenten können durch entsprechende *Tagged-Value*-Definitionen in dem jeweiligen UML-Modell für die Softwarekomponente notiert werden. Da die Art der Eigenschaften zum Zeitpunkt der Definition des UML-Profils unbekannt ist, werden die *Tagged-Value*-Definitionen nicht im virtuellen Metamodell des UML-Profils deklariert, sondern erst in einem konkreten UML-Modell einer Softwarekomponente hinzugefügt. Diese Vorgehensweise der Zuordnung von *Tagged-Value*-Definitionen zu beliebigen Modellelementen wird durch UML explizit unterstützt.

(Beispiel 22) Sei **CO1Impl** die in *Beispiel 21* definierte Softwarekomponente. Sei **XYlibrary** eine weitere Softwarekomponente und bestehe eine Abhängigkeit von **CO1Impl** zu **XYlibrary**. Sei weiterhin für **CO1Impl** die Eigenschaft **OperatingSystem** mit dem Wert „**Linux**“ definiert. Diese Situation ist in Abb.75 dargestellt.

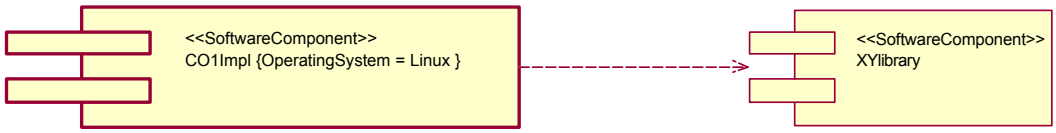


Abb. 75 Notation für Eigenschaften und Abhängigkeiten von Softwarekomponenten

5.2.8.2 Assemblage

Eine Assemblagedefinition in einem Modell ist eine Abstraktion eines realen Softwaresystems. Sie enthält alle CO-Typen, deren Instanzen zur Erfüllung des Systemzwecks des Softwaresystems benötigt werden, das durch die Assemblagedefinition im Entwurfsmodell beschrieben wird. Weiterhin gehört zu einer Assemblagedefinition die Beschreibung der initialen Konfiguration des Softwaresystems, also die Menge aller initial zu erzeugenden Instanzen der CO-Typen und der zwischen ihnen aufzubauenden initialen Bindungen.

Im UML-Profils von *CoRE$_{TATIONS}$* kann zur Notation von Assemblagedefinitionen das UML-Metamodellelement **Subsystem** benutzt werden, wobei die spezielle Semantik einer Assemblagedefinition durch eine *Stereotype*-Definition **Assembly** im UML-Profil berücksichtigt wird, die auf **Subsystem** basiert. Eine UML-**Subsystem**-Definition modelliert eine ausführbare Einheit, die sowohl logische Spezifikationselemente als auch Abstraktionen konkreter Realisierungselemente beinhalten kann. Diese Eigenschaft des UML-Metamodellelements **Subsystem** wird für die Notation einer Assemblagedefinition benötigt, wobei die logischen Spezifikationselemente dann gerade CO-Typen und die Realisierungselemente die dazugehörigen Softwarekomponenten sind.

Im Metamodell ist das Konzept **Assembly** durch eine Metaklasse **AssemblyDef** modelliert, die von der abstrakten Metaklasse **Container** abgeleitet ist. Im virtuellen Metamodell des UML-Profils wird dementsprechend eine Klasse **Assembly** definiert, die auf der Klasse **Subsystem** des UML-Metamodells basiert (vgl. Abb. 76).

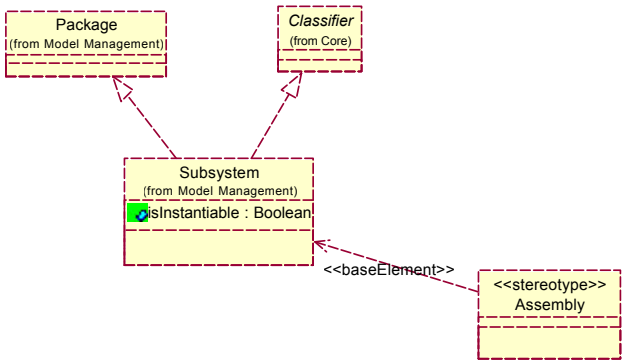


Abb. 76 Virtuelles Metamodell für Assemblage

Die zu einer Assemblagedefinition gehörigen CO-Typen können direkt als logische Spezifikationselemente der Assemblagedefinition notiert werden, da Assemblagedefinitionen auf **Subsystem** basieren. Eine zusätzli-

ches Verfahren zur Notation im UML-Profil ist zu diesem Zweck nicht notwendig. Das gleiche gilt für die Notation von Softwarekomponenten als Realisierungselemente.

5.2.8.3 Initiale COs und Initiale Bindung

Entsprechend der im Metamodell vorgesehenen Unterstützung für die initiale Konfiguration ist ein Mechanismus für die Notation der initialen COs und der zwischen ihnen aufzubauenden initialen Bindungen ebenfalls im UML-Profil von *CoRE* zu berücksichtigen und mit der Notation für Assemblagedefinitionen zu verbinden.

Zur Notation der Instanzen von CO-Typen und deren Bindungen ist das UML-Metamodellelement ***Collaboration*** geeignet. Entsprechend des UML-Standards ist eine ***Collaboration***-Definition eine Möglichkeit, um zu beschreiben, wie eine Menge von UML-Klassen und Assoziationen benutzt werden, um eine bestimmte Aufgabe zu erfüllen. Im Kontext der Notation einer Assemblagedefinition ist diese Aufgabe gerade die Herstellung der initialen Konfiguration. Die an einer solchen ***Collaboration***-Definition beteiligten UML-Klassen sind diejenigen, die für die Notation von CO-Typen verwendet werden. Weiterhin sind die zur Notation von *Port*-Definitionen verwendeten Assoziationen zwischen Klassen für CO-Typen und Interfacetypen ebenfalls Bestandteil der ***Collaboration***-Definition.

Im UML-Profil wird zur Unterstützung der initialen Konfiguration eine *Stereotype*-Definition ***initial*** definiert, die auf dem UML-Metamodellelement ***Collaboration*** basiert. Innerhalb einer solchen ***Collaboration***-Definition in einem UML-Modell wird das UML-Metamodellelement ***ClassifierRole*** zur Notation von Instanzen von CO-Typen verwendet. Weiterhin wird das UML-Metamodellelement ***AssociationRole*** zur Notation der initialen Bindungen eingesetzt. Die Vielfachheit von initialen COs zur Darstellung von Instanzmengen kann durch die Belegung des Attributes ***multiplicity*** des UML-Metamodellelements ***ClassifierRole*** notiert werden. Die entsprechend des Metamodells von *CoRE* bestehende Assoziation von Bindungen zu *Port*-Definitionen von CO-Typen wird in UML notiert, indem angegeben wird, daß die zur Notation der Bindung benutzte ***AssociationRole***-Definition auf der zur Notation der *Port*-Definition benutzten ***Association***-Definition basiert.

Im virtuellen Metamodell des UML-Profils ist somit auf der Grundlage der Klasse ***Collaboration*** des UML-Metamodells nur eine neue Klasse ***initial*** einzuführen (vgl. Abb. 77).

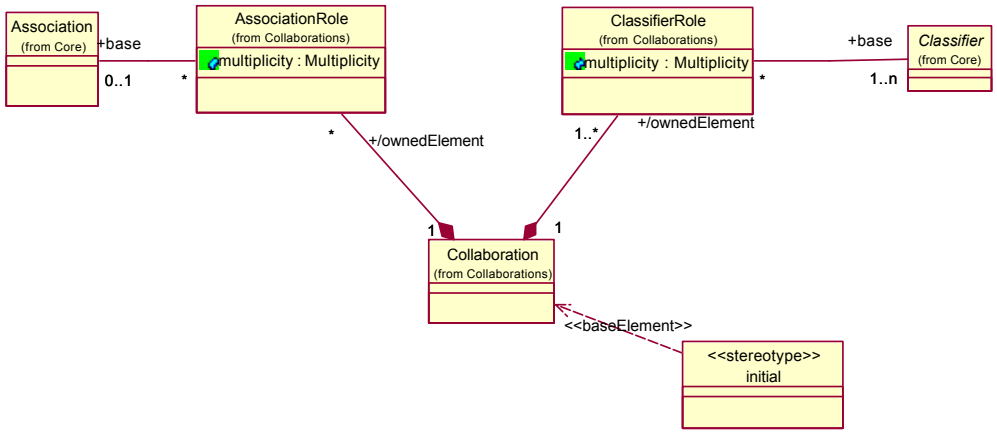


Abb. 77 Virtuelles Metamodell für initiale Konfiguration

(Beispiel 23) Sei ***A1*** eine Assemblagedefinition im Entwurfsmodell, der die CO-Typen ***CO1*** und ***CO2*** zugeordnet sind. Sei weiterhin eine Instanzmenge für ***CO1*** mit einer initialen Instanz und eine Instanzmenge von ***CO2*** mit 50 initialen Instanzen Bestandteil der initialen Konfiguration. Instanzen von ***CO1*** und ***CO2*** seien außerdem über die *Port*-Definitionen ***serve*** von ***CO1*** und ***use*** von ***CO2*** im Rahmen der initialen Bindungsdefinition ***c1*** verbunden. Dann wird diese Situation in UML wie in Abb. 78 notiert.

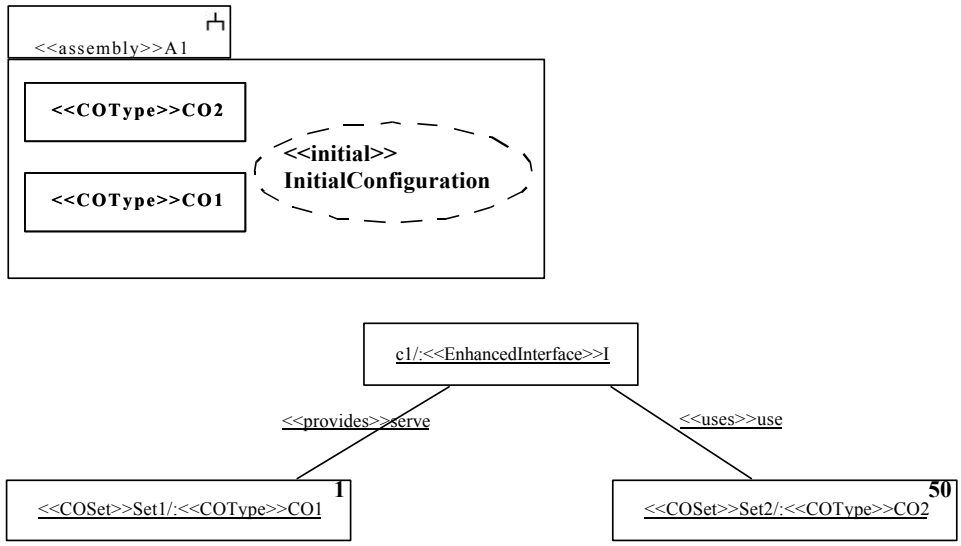


Abb. 78 UML-Notation für Assemblage und Initiale Konfiguration

5.2.9 Interaktionssicht

5.2.9.1 Kontrakttyp

Das Konzept Kontrakttyp wird verwendet, um in Entwurfsmodellen beschreiben zu können, welche Arten von Güteeigenschaften für die Aushandlung und den Aufbau eines konkreten Kontraktes, der auf einem bestimmten Kontrakttyp basiert, zur Ausführungszeit berücksichtigt werden müssen.

Im Metamodell von *CORE* ist das Konzept Kontrakttyp durch die Metaklasse ***QoSContractTypeDef*** definiert, die als Attribut eine Liste der zu einem Kontrakttyp gehörigen Dimensionen enthält. Zur Notation von Kontrakttypen wird eine *Stereotype*-Definition ***QoSContractType*** im UML-Profil vorgenommen, die auf dem UML-Metamodellelement ***class*** basiert. Die Dimensionen eines Kontrakttyps werden dementsprechend als Attribute an UML-Klassendefinitionen mit der Stereotypangabe ***QoSContractType*** notiert. Die im Metamodell geforderte Möglichkeit der Identifikation von Dimensionen innerhalb eines Kontrakttyps ist dann durch den Attributnamen gegeben. Im Metamodell von *CORE* ist zusätzlich zu der Möglichkeit der Identifikation von Dimensionen auch noch die Zuordnung einer Richtung gefordert, die beschreibt, welche Werte der Dimension bessere oder schlechtere Qualitäten im Sinne der durch die Dimension modellierten Eigenschaft darstellen. Zur Notation hierfür werden zwei *Stereotype*-Definitionen ***increasing*** und ***decreasing*** im UML-Profil eingeführt, die beide auf dem UML-Metamodellelement ***Attribute*** basieren.

Die Relationen zwischen Interfacetypen und den für sie relevanten Kontrakttypen ist im Metamodell durch eine Assoziation ***qos_contract_types*** zwischen den Metaklassen ***EnhancedInterfacedef*** und ***QoSContractTypeDef*** modelliert. Da sowohl Interfacetypen als auch Kontrakttypen als UML-Klassen notiert werden, kann zur Notation der Assoziation ***qos_contract_types*** das UML-Metamodellelement ***Association*** benutzt werden. Da weiterhin nur Assoziationen der Art ***qos_contract_types*** zwischen Interfacetypen und Kontrakttypen zulässig sind, braucht keine *Stereotype*-Definition zur Unterstützung der Notation im UML-Profil definiert werden, es wird direkt ***Association*** benutzt.

Im virtuellen Metamodell des UML-Profils werden drei Klassen ***QoSContractType***, ***increasing*** und ***decreasing*** definiert, die die oben beschriebenen *Stereotype*-Definitionen repräsentieren. ***QoSContractType*** basiert auf ***Class***, die anderen beiden basieren auf ***Attribute***. Das virtuelle Metamodell des UML-Profils für Kontrakttypen und deren Dimensionen ist in Abb. 79 dargestellt.

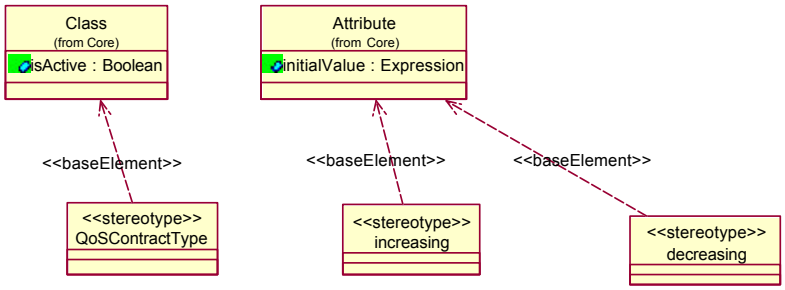


Abb. 79 Virtuelles Metamodell für QoS-Kontrakttyp

(Beispiel 24) Sei ***Performance*** ein Kontrakttyp mit den Dimensionen ***response_time*** und ***delay***. Beide Dimensionen haben den Typ ***long*** und die Richtung ***decreasing***. Sei weiterhin ***Q*** dem Interfacetyp ***I*** zugeordnet. Diese Situation wird entsprechend Abb. 80 notiert.

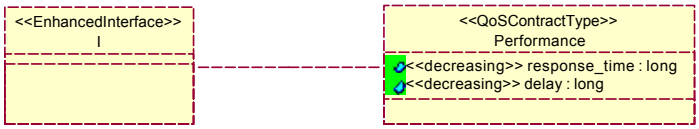


Abb. 80 Notation für QoS-Kontrakttyp

5.2.9.2 Bindungsfall, Prädikat und Bindung mit Regel

Mit der oben eingeführten Notation für Kontrakttypen sind die Voraussetzungen geschaffen, um in einem Entwurfsmodell Anforderungen von Klienten an potentielle Interaktionen im Rahmen einer Bindung notieren zu können. Solche Anforderungsspezifikationen werden als Bindungsfälle modelliert. Ein Bindungsfall enthält eine Menge von Bindungen. Für diese Bindungen werden Anforderungen über denjenigen Kontrakttypen definiert, die den der Bindung zugrundeliegenden Interfacetypen zugeordnet sind.

Analog zur Definition der Notation einer initialen Konfiguration kann hier das UML-Konzept ***Collaboration*** verwendet werden. Es sollen Instanzmengen von COs und Bindungen basierend auf ihren *Port*-Definitionen notiert werden, die allerdings zusätzlich Anforderungen formuliert als *Constraint*-Definitionen in einer geeigneten Sprache besitzen.

Zur Notation von Bindungsfällen wird eine *Stereotype*-Definition ***binding*** im UML-Profil von *CORE* eingeführt, die auf dem UML-Metamodellelement ***Collaboration*** basiert. Instanzmengen von CO-Typen werden innerhalb einer solchen ***Collaboration***-Definition mittels des UML-Metamodellelements ***ClassifierRole*** notiert, ihre Bindungen durch das Konzept ***AssociationRole***.

Entsprechend des Metamodells von *CORE* beruhen die Instanzmengendefinitionen von COs, die an einem Bindungsfall beteiligt sind, auf der Angabe von Prädikaten. Zur Notation dieser Prädikate wird eine *Stereotype*-Definition ***predicate*** eingeführt, die auf dem UML-Metamodellelement ***Attribut*** basiert. Damit können Prädikate innerhalb von CO-Typen notiert werden, für deren Darstellung im Rahmen des UML-Profils die *Stereotype*-Definition ***COType*** benutzt wird, die auf dem UML-Metamodellelement ***class*** basiert.

Zur Notation der Anforderung im Rahmen einer Bindung wird das UML-Metamodellelement ***Constraint*** verwendet. Dementsprechend wird eine *Stereotype*-Definition ***requirement*** vorgenommen, die auf dem UML-Metamodellelement ***Constraint*** basiert. Zur Notation der Sprache, in der die Anforderung definiert wird, könnte eine *Tagged-Value*-Definition an der *Stereotype*-Definition ***requirement*** erfolgen. Dies ist aber nicht notwendig, da durch UML bereits gefordert wird, daß zu einer *Constraint*-Definition immer die dafür benutzte Sprache bekannt sein muß. Eine *Tagged-Value*-Definition im Rahmen des UML-Profils wäre also redundant.

Im virtuellen Metamodell von *CORE* werden die *Stereotype*-Definitionen ***binding***, ***predicate*** und ***requirement*** als Klassendefinitionen eingeführt, wobei ***binding*** auf ***Collaboration***, ***predicate*** auf ***Attribute*** und ***requirement*** auf ***Constraint*** basiert (vgl. Abb.81).

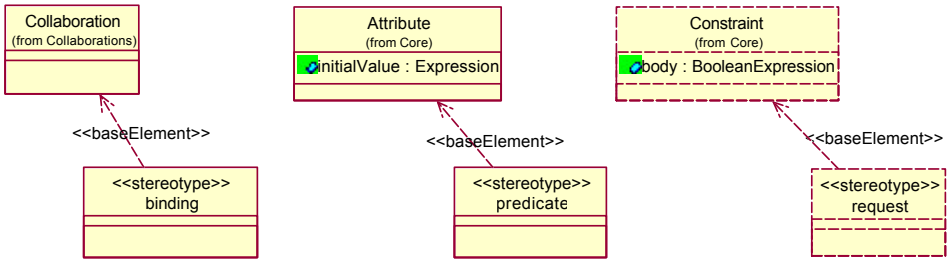


Abb. 81 Virtuelles Metamodell für Bindungsfall und Prädikat

(Beispiel 25) Sei ***b*** ein Bindungsfall, der prädikatbasierte CO-Definitionen für die CO-Typen ***CO1*** und ***CO2*** enthält. Im Falle von ***CO1*** sei das Prädikat ***p3 true***, für ***CO2*** das Prädikat ***p2 true*** und das Prädikat ***p1 false***. Sei weiterhin eine Bindung über die *Port*-Definitionen ***use*** und ***serve*** definiert, für die eine ***requirement***-Angabe "***requirement = repsonse_time < 5"*** in der Sprache OCL gefordert ist. Diese Situation wird wie in Abb.82 dargestellt.

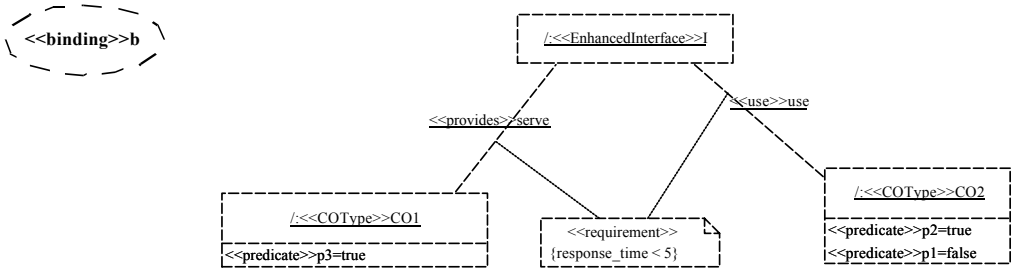


Abb. 82 Notation für Bindungsfall, Prädikat und Bindung mit Regel

5.2.10 Verwendete Stereotype-Definitionen

In Tab. 6 sind die im UML-Profil von *CORE* verwendeten *Stereotype*-Definitionen zusammengefaßt.

Konzept	***Stereotype*-Definition**	**UML-Metamodellelement**
Datentyp	entspr. CORBA-Profil	***class***
Namensraum	***Module***	***package***
Interfacetyp	***DOTInterface/EnhancedInterface***	***class***
Operation	**-**	**-**
Ausnahme	***Exception***	***class***

Tab.6 Verwendete *Stereotype*-Definitionen

Konzept	*Stereotype*-Definition	UML-Metamodellelement
Attribut	**-**	**-**
Signaltyp	***Signal***	***class***
Signalparameter	***carry***	***association***
consume	***consume***	***association***
produce	***produce***	***association***
Medium	***Medium***	***class***
Medientyp	***Mediatype***	***class***
Medienmenge	***Mediaset***	***class***
Sink	***sink***	***association***
Source	***source***	***association***
CO-Typ	***CO***	***class***
supports	***supports***	***association***
requires	***requires***	***association***
used port	***use***	***association***
provided port	***provide***	***association***
Artefakt	***Artifact***	***class***
Implementierungselement	***implelem***	***operation***
implements	***implements***	***association***
Zustandsattribut	***state***	***attribut***
Softwarekomponente	***SoftwareComponent***	***component***
Assemblage	***Assembly***	***subsystem***
Initiale Konfiguration	***initial***	***collaboration***
Kontrakttyp	***QoSContractType***	***class***
QoS-Eigenschaft (aufsteigend)	***increasing***	***attribute***
Bindungsfall	***binding***	***collaboration***
QoS-Eigenschaft (absteigend)	***decreasing***	***attribute***

Tab.6 Verwendete *Stereotype*-Definitionen

Konzept	*Stereotype*-Definition	UML-Metamodellelement
Prädikat	***predicate***	***attribute***
Bindung mit Regel	***requirement***	***constraint***

Tab.6 Verwendete *Stereotype*-Definitionen

5.3 Definition von eODL

Entsprechend der vorangegangenen Präsentation der graphischen Notation wird im folgenden korrespondierend zu den definierten Sichten des Konzeptraumes eine textuelle Darstellung der zughörigen Konzepte eingeführt. Die Definition der Grammatik erfolgt dabei unter Nutzung von EBNF (*Extended Backus Naur Form*), die gewählten Beispiele sind der Präsentation des UML-Profils entnommen. Im Gegensatz zu konventionellen Sprachdefinitionen - wie im Falle der Spezifikation von ITU-ODL im ITU-Dokument Z.130 - muß lediglich die Syntax der Sprache definiert werden, ergänzt um die Abbildung der verwendeten Sprachkonstrukte auf Metamodellelemente. *Wellformedness*-Regeln, die man in konventionellen Sprachdefinitionen üblicherweise zusätzlich findet, sind durch Verwendung von *Constraints* in der Definition des Metamodells bereits hinreichend definiert. Die im folgenden eingeführte Sprache wird als *eODL* (*extended Object Definition Language*) bezeichnet und in *Question* 2 von Studiengruppe 10 der ITU zur Standardisierung eingereicht.

5.3.1 Grundlagen der Lexik und Grammatik

Begründet in der Fundierung der Beschreibungssprache ITU-ODL auf CORBA-IDL und von *eODL* auf ITU-ODL werden für *eODL* die durch Abschnitt 3 von [OMG CORBA] festgelegten lexikalischen Konventionen übernommen. Darüber hinaus ist die vollständige Grammatik aus Abschnitt 3.4 von [OMG CORBA] Grundlage der Definition von *eODL*. Die Sprache *eODL* ist *case sensitive*. Die folgenden Bezeichner sind Schlüsselwörter in *eODL*:

abstract	**double**	**local**	**raises**	**typedef**
any	**exception**	**long**	**readonly**	**unsigned**
attribute	**enum**	**module**	**sequence**	**union**
boolean	**factory**	**native**	**short**	**ValueBase**
case	**false**	**Object**	**string**	**valuetype**
char	**fixed**	**octet**	**struct**	**void**
const	**float**	**oneway**	**supports**	**wchar**
context	**in**	**out**	**switch**	**wstring**
custom	**inout**	**private**	**true**	**default**
interface	**public**	**truncatable**	**provide**	**use**
multiple	**artifact**	**implements**	**supply**	**implemented**

Tab.7 Schlüsselwörter von eODL

by	state	provided	to	ArtifactPool
ArifactPerRequest	Singleton	UserDefined	component	realizes
assembly	contains	connection	mediatype	medium
mediaset	produce	consume	source	sink
CO	requires	supports	signal	

Tab.7 Schlüsselwörter von eODL

5.3.2 Struktursicht

5.3.2.1 Namensraum

Namensraumdefinitionen in Entwurfsmodellen werden in *eODL* durch Anwendung des CORBA-IDL-Sprachkonstruktes **module** notiert (vgl. [OMG CORBA]).

5.3.2.2 Datentyp, Interfacetyp, Ausnahme, Operation und Attribut

Dem in Kapitel3 präsentierten Konzeptraum liegt für primitive und strukturierte Datentypen sowie Ausnahmen das CORBA-IDL-Datentypsystem zugrunde. Für dieses Datentypsystem definiert [OMG CCM II] ein Metamodell, das integraler Bestandteil von *CORE*$_{CEPT}$ ist. Das UML-Profil von *CORE*$_{TATIONS}$ definiert weiterhin konkrete Ableitungsregeln zur Repräsentation von Datentypdefinitionen in Entwurfsmodellen mittels UML. Obwohl diese Regeln mit der Intension definiert wurden, CORBA-IDL-Sprachkonstrukte mittels UML darzustellen, implizieren sie auch eine kanonische Transformation der Metamodellelemente von [OMG CCM II] nach CORBA-IDL. Insofern ist die Abbildung der in einem Entwurfsmodell verwendeten Datentypen definiert und wird hier nicht im Detail diskutiert.

Operationale Interaktionselemente werden in ITU-ODL durch CORBA-IDL äquivalente Darstellungen von Operationen und Attributen notiert. Diese Notation wird für *eODL* übernommen, ebenso wie die Notation von Interfacetypen mit ausschließlich operationalen Interaktionselmenten.

5.3.2.3 Signaltyp und Signalparameter

Signalparameter von Signaltypen sind im Metamodell als Liste von Elementen des Meta-Typs ***CarryField*** an ***SignalDef*** modelliert. Konkrete Instanzen von ***CarryField*** in Entwurfsmodellen verweisen auf Instanzen von ***ValueDef***. Konsequent werden die Typen von Signalparametern durch CORBA-IDL-**<value_dcl>**-Konstrukte notiert. Für Signaltypdefinitionen wird das Konstrukt **<signal_dcl>** eingeführt. Die Signalparameter können durch Benutzung des Konstruktes **<member_list>**[1] aus CORBA-IDL definiert werden.

<signal_dcl> ::= “signal” <identifier> “{” <member_list> “}”
<member_list> ::= <member>+
<member> ::= <type_spec> <declarators> “;”

(Beispiel 26) Die in *Beispiel 10* dargestellte Situation wird mittels *eODL* folgendendermaßen notiert:

```
module M1 {
  struct S {
    string m1;
    long m2;
    };
```

1. Syntaktisch ist durch die Verwendung von **<member_list>** als Notation für Liste von ***CarryField*** die Verwendung von anderen Datentypen als *Value*-Typ zulässig. Die diese Verwendung einschränkende Semantik ist aber durch das Metamodell präzise definiert.

```
    valuetype v {
      S m1;
      };

    signal Sig1 {
      v carry_v;
    };
  };
```

5.3.2.4 Medium, Medientyp und Medienmenge

Im Metamodell sind Medientypen durch die Metaklasse ***MediaTypeDef*** modelliert, die die Eigenschaft hat, daß ihre Instanzen in einem Entwurfsmodell identifizierbar sind. Konsequenterweise werden Medientypen durch ein neues *eODL*-Konstrukt **<mediatype_dcl>** notiert.

<mediatype_dcl> ::= "mediatype" <identifier>

Das Konzept Medium besitzt eine Menge von Verweisen auf Medientypen, um die Realisierung von Medien durch Medientypen zu modellieren. In *eODL* wird hierzu ein neues Konstrukt **<medium_dcl>** eingeführt.

<medium_dcl> ::= "medium" <identifier> "(" <scoped_name> { "," <scoped_name> } * ")"

Medienmengen aggregierten eine Menge von Medien, die innerhalb einer Medienmengendefinition eindeutig identifizierbar sind. Für Medienmengen wird das neue *eODL*-Konstrukt **<mediaset_dcl>** eingeführt, die Aggregation von Medien wird über das bereits in ITU-ODL enthaltene **<member_list>**-Konstrukt notiert.

<mediaset_dcl> ::= "mediaset" <identifier> "{" <member_list> "}"

(Beispiel 27) Die in *Beispiel 13* dargestellte Situation wird mittels *eODL* folgendendermaßen notiert:

```
module M1 {
  mediatype MPEG2;
  mediatype MPEG1;
  mediatype MP3;

  medium Video ( MPEG1, MPEG2 );
  medium Audio ( MP3 );

  mediaset M {
    Video video;
    Audio audio;
  };
};
```

5.3.2.5 Erweiterter Interfacetyp, Sink-, Source-, Consume-, und Produce-Definition

Interfacetypen beinhalten eine Menge von Interaktionselementen der verschiedenen Interaktionsarten. Das bereits in ITU-ODL enthaltene Konstrukt **<interface_dcl>** ist für die Notation von operationalen Interaktionselementen geeignet und wird in *eODL* um die Konstrukte **<produce_dcl>**, **<consume_dcl>**, **<sink_dcl>** und **<source_dcl>** erweitert. Die Zuordnung der syntaktischen Elemente auf die Metamodellelmente von *CORE* ist kanonisch.

<interface_dcl> ::= <interface_header> "{" <interface_body> "}"
<interface_header> ::= ["abstract"] "interface" <identifier>[<interface_inheritance_spec>]
<interface_body> ::= <export> *
<export> ::= <type_dcl> ";"
| <const_dcl> ";"
| <except_dcl> ";"
| <attr_dcl> ";"
|<op_dcl> ";"
| <produce_dcl> ";"
| <consume_dcl> ";"

```
                | <source_dcl> ";"
                | <sink_dcl> ";"
        <produce_dcl> ::= "produce" <scoped_name> <identifier>
        <consume_dcl> ::= "consume" <scoped_name> <identifier>
        <source_dcl> ::= "source" <scoped_name> <identifier>
        <sink_dcl> ::= "sink" <scoped_name> <identifier>
```

(Beispiel 28) Die in *Beispiel 14* dargestellte Situation wird mittels *eODL* folgendendermaßen notiert:

```
module M1 {
  // Definition von Sig1, M, S und E ...
  interface I {
    produce Sig1 send_2_sig;
    produce Sig1 send_1_sig;
    consume Sig1 receive_sig;

    sink M av_in;
    source M av_out;

    void op() raises ( E );
    readonly attribute S attr;
  };
};
```

5.3.2.6 CO-Typ, Supports- und Requires-Relation

ITU-ODL erlaubt in der gegenwärtigen Version die Notation von CO-Typen sowie von *Requires*- und *Supports*-Relationen. Das Konzept des initialen Interfaces von ITU-ODL wurde durch die Definitionen der Konfigurationssicht verfeinert, so daß dieses Konzept nicht in die Definition von *eODL* übernommen wird. Stattdessen können operationale Interaktionselemente für Konfigurationszwecke direkt im Kontext von CO-Typen definiert werden.

Die Notation von CO-Typen in *eODL* wird durch die folgenden EBNF-Regeln definiert.

```
        <object_template> ::= <object_template_header>  "{" <object_template_body> "}"
        <object_template_header> ::= "CO" <identifier> [ <object_inheritance_spec> ]
        <object_inheritance_spec> ::= ":" <scoped_name> { "," <scoped_name> }*
        <object_template_export> ::= <object_export>*
        <object_export>  ::= <reqrd_interf_templates> | <suptd_interf_templates> | <export>
```

(Beispiel 29) Die in *Beispiel 15* dargestellte Situation wird mittels *eODL* folgendermaßen notiert:

```
module M1 {
  interface I {
    // ...
    };

  CO CO1 {
    requires I;
    supports I;
    void config ( in string p );
  };
};
```

5.3.3 Konfigurationssicht

5.3.3.1 Port-, Used- und Provided-Port-Definition

Port-Definitionen werden im Kontext von CO-Typen definiert, dementsprechend werden die EBNF-Regeln für **<object_export>** um syntaktische Konstrukte für die Darstellung von *Used*- und *Provided-Port*-Definitionen erweitert.

```
<object_export>  ::= <reqrd_interf_templates>
                   | <suptd_interf_templates>
                   | <export>
                   | <use_dcl> ";"
                   | <provide_dcl> ";"
<use_dcl> ::= "use" [ "multiple"] <scoped_name> <identifier>
<provide_dcl> ::= "provide" [ "multiple"] <scoped_name> <identifier>
```

(Beispiel 30) Die in *Beispiel 16* dargestellte Situation, erweitert um eine auf ***I*** basierende *Used-Port*-Definition im Kontext von ***CO1***, wird mittels *eODL* folgendendermaßen notiert:

```
module M1 {
  CO CO1 {
    supports I;
    requires I;
    provide I serve;
    use multiple I user;
  };
};
```

5.3.4 Implementierungssicht

5.3.4.1 Artefakt und Implementierungselement

Artefakte sind konzeptionell Abstraktionen von programmiersprachlichen Konstrukten, die das spezifische Verhalten von CO-Typen realisieren. Diese Relation wird durch *Implements*-Relationen zwischen CO-Typen und Artefakten in Entwurfsmodellen ausgedrückt. Artefakte enthalten Implementierungselemente, die wiederum Interaktionselemente von Interfaces realisieren. Im Kontext eines Artefakts definiert ein Implementierungselement also, welches konkrete Interaktionselement eines Interfacetyps im *Supply*- bzw. *Use*-Fall realisiert wird.

Die Syntax für Artefakt- und Implementierungselementdefinitionen ist durch die folgenden EBNF-Regeln fixiert:

```
<artifact> ::= <artifact_dcl>
             | <artifact_forward_dcl>
<artifact_forward_dcl> ::= "artifact" <identifier>
<artifact_dcl> ::= <artifact_header> "{" <artifact_body> "}"
<artifact_header> ::= "artifact" <identifier> [ <artifact_inheritance_spec> ]
<artifact_inheritance_spec> ::= ":" <scoped_name> { "," <scoped_name> }*
<artifact_body> ::= [ <impl_elem_dcl>* ]
<impl_elem_dcl> ::= <identifier> "implements" <impl_case_dcl> <scoped_name> ";"
<impl_case_dcl> ::= "supply" | "use"
```

Entsprechend diesen EBNF-Regeln ist sowohl die Vorwärtsdeklaration (**<artifact_forward_dcl>**) als auch die Spezifikation von Basisartefakten (**<artifact_inheritance_spec>**) möglich. Durch die Definition der EBNF-Regel für **<artifact_body>** ist zur Unterstützung des iterativen Entwicklungsprozesses die Spezifikation von leeren Artefakten (d.h. Artefakte ohne Implementierungselemente) ermöglicht.

(Beispiel 31) Die in *Beispiel 18* dargestellte Situation wird mittels *eODL* folgendermaßen notiert:

```
module M1
{
  artifact A1
  {
    attr_impl implements supply I::attr;
    op_impl implements supply I::op;
    send_1_sig_impl implements supply I::send_1_sig;
    send_2_sig_impl implements supply I::send_2_sig;
    receive_sig_impl implements supply I::receive_sig;
```

```
    av_out_impl implements supply I::av_out;
    av_in_impl implements supply I::av_in;
  };

  artifact A2
  {
    receive_1_sig_impl implements use I::send_1_sig;
    receive_2_sig_impl implements use I::send_2_sig;
    send_sig_impl implements use I::receive_sig;
    av_out_impl implements use I::av_in;
    av_in_impl implements use I::av_out;
  };
};
```

5.3.4.2 Implements-Relation

Die Definition von *Implements*-Relationen erfolgt im Kontext von CO-Typen, um die Realisierung deren Verhaltens durch Artefakte zu spezifizieren. Dementsprechend werden die EBNF-Regeln für **<object_export>** um syntaktische Konstrukte für die Darstellung dieser Relationen erweitert.

```
<object_export> ::= <reqrd_interf_templates>
                  | <suptd_interf_templates>
                  | <export>
                  | <use_dcl>
                  | <provide_dcl>
                  | <implements_dcl>
<implements_dcl> ::= "implemented" "by" <scoped_name> { "," <scoped_name>* } ";"
```

Mit Hilfe dieser syntaktischen Konstrukte und jenen für Artefakte selbst ist die Zuordnung von Interaktionselementen zu Implementierungselementen von Artefakten im Kontext eines CO-Typs definiert.

(Beispiel 32) Die in *Beispiel 17* dargestellte Situation wird mittels *eODL* folgendermaßen notiert:

```
module M1 {
  artifact A1;
  artifact A2;
  artifact A3;

  CO CO1 {
    implemented by A1, A2, A3;
  };
};
```

5.3.4.3 Zustandsattribut

Die Realisierung des Verhaltens von CO-Typen durch Artefakte erfordert i.allg. den Zugriff auf Zustandsinformationen von COs durch konkrete Implementierungselemente. Zustandsinformationen von CO-Typen können entsprechend Kapitel 3 in Form von CORBA-IDL-Datentypen repräsentiert sein. Im Kontext eines CO-Typs werden zum einen Zustandsattribute spezifiziert sowie die Nutzung dieser durch Realisierungen von Implementierungselementen definiert. Zur Definition der zu verwendenden syntaktischen Konstrukte wird die EBNF-Regel für **object_export** erweitert:

```
<object_export> ::= <reqrd_interf_templates>
                  | <suptd_interf_templates>
                  | <export>
                  | <use_dcl>
                  | <provide_dcl>
                  | <implements_dcl>
                  | <state_def_dcl>
<state_def_dcl> ::= "state" <scoped_name> [ "provided" "to" "(" <provided_to_dcl> ")" ] ";"
<provided_to_dcl> ::= <scoped_name> { "," <scoped_name> }*
```

(Beispiel 33) Zur Illustration seien **Foo** und **Bar** zwei CO-Typen und **I** ein Interfacetyp im Namensraum **M1**. **I** enthalte eine Operation **op**, die durch das Implementierungselement **op** im Artefakt **A** implementiert wird. Der CO-Typ **Bar** bietet das Interface **I** an, und wird durch **A** realisiert. Er ein Zustandsattribut **bar_state** vom Typ **S**, wobei **S** eine Strukturdefinition mit einem *Member*-Element **the_foo** vom Typ **Foo** ist. Dieses Zustandsattribut wird durch das Implementierungselement **op** des Artefakts **A** benutzt.

```
module M1 {
  interface I {
    void op();
  };
  artifact A {
    op implements supply I::op;
  };
  CO Foo { /* ... */ };
  struct S {
    Foo the_foo;
  };
  CO Bar {
    implemented by A;
    state S bar_state provided to ( A::op );
  };
};
```

5.3.4.4 Instanziierungsmuster

Im Kontext eines CO-Typs können für Artefakte Instanziierungsmuster (*Instantiation Policies*) angegeben werden. Diese definieren, auf welche Weise und zu welchen Zeitpunkten Instanzen von Artefakten zur Implementierung des Verhaltens von Interaktionselementen erzeugt werden. Instanziierungsmuster werden in *eODL* durch eine Erweiterung der Definition von **<implements_dcl>** spezifiziert.

```
<implements_dcl> ::= "implemented" "by" <scoped_name_with_policy>
                     { "," <scoped_name_with_policy>* } ";"
<scoped_name_with_policy> ::= <scoped_name> [ "with" <instantiation_policy_dcl> ]
<instantiation_policy_dcl> ::= "ArtifactPool"
                     | "ArtifactPerRequest"
                     | "Singleton"
                     | "UserDefined"
```

Die Angabe eines Instanziierungsmusters ist in *eODL* optional, falls in einer *eODL* Definition keine solche Regel angegeben wird, gilt **UserDefined** als *Default*-Fall angenommen.

(Beispiel 34) Die in *Beispiel 33* dargestellte Situation sei um das Instanziierungsmuster **ArtifactPool** für das Artefakt **A** erweitert:

```
module M1 {
  CO Bar {
    implemented by A with ArtifactPool;
  };
};
```

5.3.5 Deployment-Sicht

5.3.5.1 Softwarekomponenten und Realize-Relation

Entsprechend den Definitionen des Metamodells können CO-Typen in Softwarekomponenten enthalten sein. Diese Relationen drücken aus, daß die produzierten Codefragmente der Repräsentationen der CO-Typen Bestandteil einer Softwarekomponente sind. Entsprechend der semantischen Fundierung des Konzeptes Softwarekomponente werden in *eODL* die folgenden EBNF-Regeln zur syntaktischen Definition der Relationen zwischen CO-Typen und Softwarekomponenten eingeführt:

```
<component_dcl> ::= "component" <identifier> "realizes" ("( <realizes_dcl> ")"
<relalizes_dcl> ::= <scoped_name> { "," <scoped_name> > }*
```

(Beispiel 35) Eine Softwarekomponente **BarFoo**, die die in *Beispiel 33* definierten CO-Typen **Bar** und **Foo** realisiert, wird mittels *eODL* folgendermaßen definiert:

```
component BarFoo realizes ( M1::Bar, M1::Foo );
```

5.3.5.2 Assemblage, initiale COs und initiale Bindung

Assemblagedefinitionen sind entsprechend des Metamodells von *CORE* Container für initiale COs und deren initiale Bindungen. Sie beinhalten außerdem eine Liste der zu der Assemblagedefinition gehörigen CO-Typen. Die initialen COs sind in Form von Instanzmengen modelliert, für die die Zahl der initialen Instanzen angegeben ist. Initiale Bindungen beziehen sich auf *Port*-Definitionen der zu den Instanzmengen gehörigen CO-Typen.

In *eODL* werden die folgenden EBNF-Regeln zur Notation von Assemblagedefinitionen eingeführt:

```
<assembly_dcl>::= "assembly" <identifier> <assembly_header> "{"<assembly_body>"}"
<assembly_header>::="contains (" <contained_co_dcl> ")"
<contained_co_dcl>::=<scoped_name> { "," <scoped_name> }*
<assembly_body>::=<assembly_export>*
<assembly_export>::=<co_set_dcl>";"
        |<connection_dcl>";"
<co_set_dcl>::=<scoped_name><identifier>"("<integer_literal> ")"
<connection_dcl>::="connection"<identifier>"("<port_of_co_dcls>* ")"
<port_of_co_dcls>::=<port_of_co_dcl>
        | <port_of_co_dcl>","<port_of_co_dcls>
<port_of_co_dcl>::=<identifier>":"<identifier>
```

(Beispiel 36) Sei **A1** die Assemblagedefinition aus *Beispiel 23*. Dann wird **A1** in *eODL* folgendermaßen notiert:

```
assembly A1 contains ( CO1, CO2 ) {
  CO1 Set1 ( 1 );
  CO2 Set2 ( 50 );
  connection c1 (Set1:serve, Set2:use);
};
```

5.3.6 Interaktionssicht

Im Kontext der Diskussion des Lösungsansatzes und der Definition des Metamodells wurde auf die Arbeiten zur Beschreibungssprache *Quality Modeling Language* (QML, [FK 98]) verwiesen. Die Autoren von [FK 98] haben für diese Beschreibungssprache eine konkrete Syntax definiert. Während bei der Definition des UML-Profils von *CORETATIONS* eine neue graphische Syntax für die Konzepte der Interaktionssicht eingeführt werden konnte, erfordert die Definition der konkreten Syntax von *eODL* für die Interaktionssicht eine Integration von *eODL* mit der Syntax von QML. Da nicht nur diese Integration, sondern auch die semantische Fundierung - die Integration der Konzepte von QML mit dem Metamodell von *CORE* - gegenwärtig in internationalen und nationalen Projekten [vHalt00][ComPoTel00] studiert wird, wurde im Kontext dieser Arbeit auf die Definition einer konkreten Syntax für die Interaktionssicht verzichtet.

5.4 Diskussion

Die in diesem Kapitel vorgenommene Definition der Notation für *CORE* hat einerseits gezeigt, daß sich die im Konzeptraum eingeführten Konzepte sowohl textuell als auch graphisch darstellen lassen. Andererseits wurde demonstriert, daß die Definition einer Notation einen vergleichsweise kleinen Aufwand bei der Aus-

arbeitung der Entwicklungstechniken erfordert, wenn zuvor die Konzepte in einem wohldefinierten Metamodell präzisiert wurden. Das hier vorgestellte und demonstrierte Verfahren zur Notationsdefinition ist unabhängig von der Anwendungsdomäne Telekommunikation und stellt somit einen methodischen Fortschritt bei der Entwicklung von Notationen zur Unterstützung der Softwareentwicklung i.allg. dar. So läßt sich das Verfahren innerhalb von Standardisierungsgremien wie OMG oder ITU, die mit der Entwicklung von Notationen befaßt sind, etablieren, um die Standardisierungsarbeiten effizienter und mit qualitativ höherwertigen Ergebnissen durchzuführen. Ein entsprechender Vorschlag wurde der ITU bereits unterbreitet. Konkret werden die ITU-Sprachen ITU-ODL und ITU-DCL auf einem gemeinsamen Metamodell basieren, während konkrete Notationen nur noch in Form von Anhängen dem jeweiligen Standard hinzugefügt werden sollen. Der Hauptvorteil ist darin zu sehen, daß man sich bei der Sprachdefinition auf die Fundierung der Konzepte konzentrieren kann und die konkrete Notation ein Nebenprodukt ist, welches sich in verschiedenen Varianten leicht aus dem Metamodell herleiten läßt. Dabei können sowohl bestehende Notationen verwendet als auch neue definiert werden. Ebenfalls elegant gelöst ist das Problem eines Modellaustausches zwischen verschiedenen Notationen, der nunmehr über das gemeinsame Metamodell erfolgen kann.

KAPITEL 6

Resümee und Ausblick

In diesem Band wurde die objektorientierte Modellierung als eine der in *CORE* zusammengefaßten Entwicklungstechniken für die Entwicklung von verteilten Telekommunikationssoftwaresystemen präzise definiert. Diese Definition erfolgte durch die objekt-orientierte Konstruktion eines Metamodells, in dem alle Konzepte und deren Beziehungen erfaßt wurden, die für die Modellbildung in der Entwurfsphase innerhalb von *CORE* einsetzbar sind. Als Technologie zur Definition und Realisierung wurde *Meta Object Facility* eingesetzt. Ausgangspunkt dieser Konstruktion war eine Analyse internationaler Standards und wissenschaftlicher Arbeiten. Aus diesen Ansätzen wurden Konzepte und Beziehungen hergeleitet, die zum Entwurf von verteilten Telekommunikationssoftwaresystemen bezüglich der in [CoRE I] diskutierten Anforderungen an solche Systeme geeignet sind. Die Konstruktion dieses Konzeptraumes ist zentrales Element der von *CORE* realisierten Integration der Entwicklungstechniken objektorientierte Modellierung und Einsatz von Komponentenarchitekturen. Diese Integration wird in *CORE* durch die Definition von Ableitungsregeln zur Ableitung von Softwarekomponenten aus objektorientierten Entwurfsmodellen erreicht. Die Regeln sind auf der Basis der im konstruierten Konzeptraum definierten Konzepte und deren Beziehungen formuliert, ihre Darstellung erfolgt gemeinsam mit der Konzeption einer erweiterten Komponentenarchitektur basierend auf CORBA in [CoRE III].

Konzeptionelle Grundlagen des hier konstruierten Konzeptraumes sind die Ansätze von *Reference Model for Open Distributed Processing (RM-ODP)*, *TINA Computational Modeling Concepts* und CORBA *Components*. In $CORE_{CEPT}$ wurden u.a. die zentralen Konzepte CO-Typ mit *Port*-Definitionen, Interfacetyp mit Interaktionselementen, Artefakt mit Implementierungselementen und Instanziierungsmustern, Assemblage mit initialer Konfiguration, Softwarekomponente sowie Bindung mit Regel eingeführt. Das Konzept CO-Typ wurde hinsichtlich seiner Basiseigenschaften aus RM-ODP übernommen und mit den in CORBA *Components* präzisierten Konfigurationsaspekten des dort definierten Meta-Typs **component** integriert. Diese Integration wurde durch die Einführung des Konzeptes *Port*-Definition erreicht, das den Austausch von Interfacereferenzen zwischen COs beschreibt. Konkrete *Port*-Definitionen in Entwurfsmodellen können unabhängig von den Arten der in den dazugehörigen Interfacetypen aggregierten Interaktionselemente vorgenommen werden - damit wird die Modellbildung und die Darstellung der Entwurfsmodelle gegenüber CORBA

Components wesentlich vereinfacht und zudem die dort nicht unterstützte Interaktionsart *Continuous Media* Interaktion eingeschlossen. Diese Vereinfachung wurde durch die Erweiterung des Konzeptes Interfacetyp aus RM-ODP und CORBA *Components* erreicht - Interfacetypen in $CoRE_{CEPT}$ werden nicht nach der Art ihrer Interaktionselemente (operationale, Signal- und *Continuous-Media*-Interaktionselemente) unterschieden. In $CoRE_{CEPT}$ wird das Konzept Interfacetyp als Definition eines Interaktionskontexts verstanden und bezüglich der Interaktionsarten homogen definiert. Die in RM-ODP enthaltenen Konzepte erlauben keine Modellierung der internen Struktur und des internen Verhaltens von CO-Typen. Um jedoch die objektorientierte Modellierung mit dem Einsatz von Komponentenarchitekturen zu integrieren, ist eine diesbezügliche Beschreibung notwendig. In $CoRE_{CEPT}$ wurden dazu die Konzepte Artefakt und Implementierungselement aufgenommen. Artefakt beschreibt dabei programmiersprachliche Konstrukte, die das spezifische Verhalten von COs realisieren. Die Zuordnung von Interaktionselementen auf die sie realisierenden Artefakte erfolgt unter Verwendung des Konzeptes Implementierungselement. Der Lebenszyklus von Artefakten wird durch die Verwendung des Konzeptes Implementierungselement in Verbindung mit Instanziierungsmustern in Entwurfsmodellen festgelegt. Das Konzept Softwarekomponente, u.a. durch [Szy99] präzisiert, wurde in $CoRE_{CEPT}$ integriert und gestattet die Zuordnung von Elementen des Entwurfsmodells (i.allg. CO-Typen) zu konkreten Softwarekomponenten. Die Anwendung dieses Konzeptes in Entwurfsmodellen beschreibt diejenigen Softwarekomponenten und die enthaltenen Codemodule, die durch $CoRE_{MAP}$ automatisch erzeugt werden. Die zu einem Telekommunikationssoftwaresystem gehörigen CO-Typen, Softwarekomponenten, deren Eigenschaften und die zum Zeitpunkt der Inbetriebnahme des Systems zu realisierende initiale Konfiguration wird durch die Verwendung der Konzepte Assemblage und initiale Konfiguration bereits in Entwurfsmodellen erfaßt. Die Verwendung dieser Konzepte ermöglicht die Erfassung konkreter Einsatzbedingungen eines zu entwickelnden Softwaresystems und die dementsprechende Zusammenstellung und Konfiguration seiner Komponenten. Diese Konzepte der *Deployment*-Sicht von *CoRE* waren u.a. Grundlage der Untersuchungen im EURESCOM Projekt P924 [EUP924] bezüglich der automatisierten Überführung eines entwickelten Softwaresystems in die Einsatzphase. Die Erfassung nicht-funktionaler Eigenschaften und Anforderungen eines Telekommunikationssoftwaresystems in Entwurfsmodellen wird durch das Konzept Bindung mit Regel unter Benutzung des Konzeptes Kontrakttyp ermöglicht. $CoRE_{CEPT}$ realisiert dabei eine fallbasierte Spezifikation dieser nicht-funktionalen Eigenschaften und Anforderungen.

$CoRE_{CEPT}$ stellt also Konzepte bereit, die zum Entwurf von verteilten Telekommunikationssoftwaresystemen geeignet sind, die die in [CoRE I] diskutierten Anforderungen an solche Systeme erfüllen.

- Die Offenheit eines Softwaresystems und dessen Komponenten wird in *CoRE* dadurch erreicht, daß ausschließlich wohldefinierte Interfaces zur Interaktion zwischen diesen Komponenten benutzt werden. Die in $CoRE_{MAP}$ enthaltenen Ableitungsregeln gestatten die Erzeugung von mindestens quellenportablen Softwarekomponenten. Die Bereitstellung von Entwurfsinformationen über Softwarekomponenten erfolgt durch Repositorien, die auf *Meta Object Facility* basieren bzw. XML-Dokumente auf der Basis von XMI.
- Die konzeptionelle Offenheit von $CoRE_{CEPT}$ ist durch die objektorientierte Konstruktion auf der Grundlage von *Meta Object Facility* sichergestellt. Zusätzliche Konzepte können durch Verfeinerung der in $CoRE_{CEPT}$ enthaltenen Konzepte bzw. durch Definition von Beziehungen zu bereits existierenden Konzepten berücksichtigt werden. Als Beispiel sei hier der Standardisierungsprozeß für ITU-ODL und ITU-DCL [ITUT Z.130] angeführt - Das Metamodell von $CoRE_{CEPT}$ wird in diesem Prozeß als Basis verwendet und durch Konzepte zur detaillierten Beschreibung von Ausführungsumgebungen erweitert.
- Gütebeschreibung und Garantie ist eine Anforderung, die durch Konzepte der Interaktionssicht von $CoRE_{CEPT}$ in Entwurfsmodellen hinsichtlich der Gütebeschreibung realisiert werden kann. Die Umsetzung der Gütebeschreibung und die Unterstützung der Gütegarantie ist hingegen eine Eigenschaft der Komponentenarchitektur, die als Ausführungsumgebung eines entwickelten Softwaresystems genutzt wird, in *CoRE* zeichnen $CoRE_{WARE}$ und $CoRE_{MAP}$ dafür verantwortlich.
- Flexible Skalierbarkeit wird in $CoRE_{CEPT}$ insbesondere durch die Festlegung von Instanziierungsmustern in Entwurfsmodellen erreicht. In einem iterativen Prozeß können während der Entwicklung einer Soft-

warekomponente durch Angabe verschiedener Instanziierungsmuster für Artefakte in Entwurfsmodellen so lange Experimente durchgeführt werden, bis entsprechende *Performance*-Analysen den Skalierbarkeitsanforderungen an die zu entwickelnde Softwarekomponente gerecht werden. Ein solches iteratives Vorgehen wird gerade durch die in *CoRE* zusammengefaßten Entwicklungstechniken ermöglicht.

- Flexible Adaptierbarkeit wird durch konsequente Verwendung objektorientierter Paradigmen sowie durch die konzeptionelle Trennung zwischen den Konzepten CO-Typ und Artefakt erreicht. Letzteres gestattet die potentiell getrennte Wiederverwendung der strukturellen Aspekte eines Entwurfsmodells bezüglich der Interaktion sowie der Aspekte der Realisierung von CO-Typen durch Artefakte. So können Artefaktdefinitionen zusammen mit ihren programmiersprachlichen Realisierungen im Kontext verschiedener CO-Typen wiederverwendet werden.
- Die Unterstützung von *Continuous-Media*-Interaktionselementen in Entwurfsmodellen ist in $CoRE_{CEPT}$ durch geeignete Konzepte berücksichtigt. Darüber hinaus können solche Interaktionselemente gemeinsam mit Interaktionselementen anderer Interaktionsarten im Kontext ein und desselben Interfacetyps definiert werden.
- Die Unterstützung der flexiblen Integration von Softwarekomponenten (eine zentrale Anforderung an die eingesetzte Komponentenarchitektur) wird in *CoRE* bereits in der Entwurfsphase vorbereitet. Abstraktionen unterschiedlicher Softwarekomponenten gemeinsam mit ihren Eigenschaften sind Bestandteil der Spezifikation eines Softwaresystems und dienen der Parametrisierung der Ableitungsregeln von $CoRE_{MAP}$.
- Kurze Entwicklungszeiten für Softwaresysteme werden durch die integrative Anwendung der in *CoRE* zusammengefaßten Entwicklungstechniken im Kontext von Iterationen erreicht. Die vollständige Implementierung von Entwicklungswerkzeugen, die die Entwicklungstechniken von *CoRE* automatisieren, ist bereits realisiert [BK 01a].

Zur Darstellung von Entwurfsmodellen, die auf der Basis von $CoRE_{CEPT}$ definiert werden, sind hier exemplarisch eine graphische und eine textuelle Notation eingeführt worden. Damit ist die Machbarkeit des Ansatzes der Trennung von konzeptioneller Fundierung und nachfolgend vorgenommener Notationsdefinitionen nachgewiesen. Die Entwicklungstechnik objektorientierte Modellierung konnte diesem Ansatz folgend vereinfacht definiert werden - die in der Vergangenheit üblichen Diskussionen über die Eignung oder Nichteignung von konkreten Darstellungsmitteln für spezifische Konzepte wurden obsolet.

Die Verwendung von *Unified Modeling Language (UML)* hat sich bei der Definition der graphischen Notation aus mehreren Gründen als vielversprechender Ansatz erwiesen:

- UML hat einen hohen Verbreitungsgrad erlangt - Entwurfsmodelle, die mittels UML notiert sind, können von einer großen Gruppe von Entwicklern hinsichtlich der modellierten Kernaspekte *ad hoc* verstanden werden,
- Die objektorientierten Paradigmen, die zur Konstruktion von $CoRE_{CEPT}$ verwendet wurden, sind integraler Bestandteil von UML - UML bietet auf dieser Basis einen standardisierten Erweiterungsmechanismus,
- UML bietet geeignete Ausdrucksmittel für eine Vielzahl in ihrer Natur verschiedener Sachverhalte - so können sowohl strukturelle als auch Verhaltensaspekte objektorientiert notiert werden,
- der hohe Verbreitungsgrad von UML ist nicht zuletzt in einer breiten Werkzeugunterstützung begründet.

Durch die Definition der textuellen Notation *eODL* wurde die Beschreibungssprache ITU-ODL erweitert. Es können nunmehr alle[1] Aspekte von verteilten Telekommunikationssoftwaresystemen, die in $CoRE_{CEPT}$ zusammengefaßt sind, auch textuell notiert werden. Auf der Basis der Realisierung des Metamodells von $CoRE_{CEPT}$ wurden Entwicklungswerkzeuge durch den Autor realisiert, die die Überführung von in UML bzw. *eODL* dargestellten Entwurfsmodellen oder Teilmodellen ineinander gestatten.

1. Einschränkungen gelten nur bzgl. der Interaktionssicht wie in Abschnitt 5.3.6 dargestellt.

Der Konzeptraum von *CoRE* sowie die in $CoRE_{TATIONS}$ eingeführten Notationen wurden im Rahmen internationaler Konferenzen zur Softwareentwicklung für verteilte Systeme (vgl. z.B. [BHK 01], [FKB 00] und [BKH 00]) und in Standardisierungsgremien (vgl. z.B. [BK 00b], [BK01b]) begleitend zur Entwicklung von *CoRE* mehrfach vorgestellt. Die daraus entstandenen Diskussionen mit zahlreichen Experten haben im wesentlichen die prinzipielle Übertragbarkeit auf andere Bereiche der Softwareentwicklung gezeigt, woraus sich konkrete Untersuchungsgegenstände für weitere Arbeiten ergeben.

Der mögliche Einfluß von $CoRE_{CEPT}$ und $CoRE_{TATIONS}$ auf aktuelle oder in Vorbereitung befindliche Standardisierungsprozesse sowie der Einsatz und die Weiterentwicklung von *CoRE* in Industrieprojekten wurden bereits in [CoRE I], Kapitel4 dargestellt. An dieser Stelle seien daher nur einige konkrete Aspekte der Werkzeugunterstützung und Weiterentwicklung von $CoRE_{CEPT}$ und $CoRE_{TATIONS}$ explizit hervorgehoben:

- Die Realisierung von $CoRE_{CEPT}$ in der derzeitigen Implementierung von *CoRE* erfolgte durch die Bereitstellung einer C++-basierten Klassenbibliothek. Zwischenzeitlich sind jedoch Werkzeuge kommerziell verfügbar, die den Standard *Meta Object Facility* implementieren und somit die Überführung von Metamodellen in entsprechend erzeugte Repositorien gestatten. Um die Flexibilität von $CoRE_{CEPT}$ weiter zu erhöhen, ist der Einsatz solcher Werkzeuge anzustreben. Damit ist dann der Zugriff auf Entwurfsinformationen von Softwarekomponenten während deren Einsatz erheblich einfacher, da durch *Meta Object Facility* geeignete CORBA-IDL-Interfacedefinitionen erzeugt werden.
- Die gegenwärtige Implementierung der graphischen Notation von $CoRE_{TATIONS}$ ist als direkte Anbindung an das UML-Werkzeug Rational Rose [RationalRose] vorgenommen worden. In einem nächsten Entwicklungsschritt sollten weitere UML-Werkzeuge integriert werden. Eine geeignete Technologie hierfür bietet XMI.
- UML liegt gegenwärtig in der Version 1.3 vor. Gegenwärtige Standardisierungsaktivitäten innerhalb der *Object Management Group* haben eine Sprachrevision zum Ziel, die den Umfang von UML wesentlich verändert. Dadurch notwendig werdende Anpassungen bzw. weitere Vereinfachungen der Notationen von $CoRE_{TATIONS}$ berücksichtigt werden.

Das Potential von *CoRE* wird durch die Integration von $CoRE_{CEPT}$ und $CoRE_{TATIONS}$ mit den Ableitungsregeln für die automatische Ableitung von Softwarekomponenten aus Entwurfsmodellen $CoRE_{MAP}$ und der Komponentenarchitektur $CoRE_{WARE}$ entfaltet. $CoRE_{MAP}$ *UND* $CoRE_{WARE}$ sind zentrale Bestandteile der Ausführungen in [CoRE III].

REFERENZEN

[ANSIC++] ISO/IEC 14882:1998: "*Programming languages - C++*", ISO '98

[BF 01] Born, Fischer: "*Object Definition Language*" Telektronikk 04/2001, Kjeller, Norway '01

[BFL+ 99] Born, Fischer, Löwis, Krüger, Ulbricht: "*Service Composition in a TINA Enviroment*", Proceedings of the TINA 1999 Conference, Oahu, USA '99

[BH 98] Born, Hoffmann: "*An Object-Oriented Design Methodology for Distributed Systems*", Proceedings of the Technology of Object-Oriented Languages and Systems 1998 (TOOLS Pacific 1998) Conference, Melbourne, Australia '98

[BH 98a] Born, Hoffmann: "*Advanced Distribution and Configuration Support for Distributed Applications*" Proceedings of the Workshop on Distributed Object Technologies for Telecoms Networks (DOT'98), Heidelberg, Germany '98

[BHK 01] Born, Holz, Kath: "*Manufacturing Software Components from Object-Oriented Design Models*", Proceedings of the 5th International Enterprise Distributed Object Computing Conference (EDOC 2001), Seattle, USA '01

[BHL+ 98] Born, Hoffmann, Li, Schieferdecker: "*Applying a Framework Approach with Validation to the Design of Telecommunication Services*": Proceedings of the International Conference on Communication Technologies (ICCT'98), Beijing, China '98

[BHL+ 99] Born, Hoffmann, Li, Schieferdecker: "*Using Formal Methods for the Design of Telecommunication Services*" Proceedings of the Conference on Formal Methods for Object Oriented Distributed Systems (FMOODS) '99, Florence, Italy '99

[BHS+99] Born, Hoffmann, Schieferdecker, Vassiliou-Gioles, Winkler: "*Performance Testing of a TINA Platform*" : Proceedings of the TINA 1999 Conference, Oahu, USA '99

[BHW98] Born, Hoffmann, Winkler: "*SDL Enhancements and Integration into the Design-Lifecycle of Telecommunication Services*", Proceedings of the International Conference on Communication Technologies (ICCT'98), Beijing, China '98

[BHW 98a] Born, Hoffmann, Winkler: "*The ITU-ODL to C++ Mapping and its Integration into an SDL based Design-Methodology for Telecommunication Services*", Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC (SAM´98), Berlin, Germany '98

[Bitk 00] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien: "*Informationstechnik und Telekommunikation im Dauerhoch*", Bitkom-Presseinformation, http://www.bitkom.org/presse/pr230200.htm, BitKom '00

[Bitk 00a] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien: "*Wege in die Informationsgesellschaft - Status quo und Perspektiven Deutschlands im internationalen Vergleich*", BitKom '00

[BK 00] Born, Kath: "*From TINA-ODL Towards a Component Oriented Design Method*". Proceedings of the TINA 2000 Conference, Paris, France '00

[BK 00a] Born, Kath: "*Code Generation for Component based Telecommunication Service Development*", Proceedings of the SoftCom 2000 Conference, Split, Croatia '00

[BK 00b] Born, Kath: "*Customizing UML for Component Design*", Proceedings of the 1st OMG Workshop: UML In The .com Enterprise - Modeling CORBA, Components, XML/XMI And Metadata, Palm Springs, USA '00

[BK 01] Born, Kath: "*Distributed Applications: From Models to Components*", EURESCOM Workshop on Middleware in Telecommunications, Kjeller, Norway '01

[BK 01a] Born, Kath: "*The DOT Profile*", http://www.dot-profile.de/

[BK 01b] Born, Kath: "*Distributed Applications: From Models to Components*", Presentation at the Object Management Group TC Meeting, document number telecom/01-03-02, Irvine, USA '01

[BKH 00] Born, Kath, Holz: "*A UML Profile for Integrated Design and Development of Distributed Applications*", Proceedings of the Technology of Object-Oriented Languages and Systems 2000 (TOOLS Pacific 2000) Conference, Sydney, Australia '00

[BKvH00] Born, Kath, v. Halteren: „*Modeling and Runtime Support for Quality of Service in Distributed Component Platforms*", Work in Progress Paper at 11th Annual IFIP/IEEE International Workshop on "Distributed Systems: Operations and Management", Austin, USA '00

[Box99] Box: "*COM - The Component Object Model*", Addison-Wesley '98

[BMBF 00] Bundesministerium für Bildung und Forschung: "*Analyse und Evaluation der Softwareentwicklung in Deutschland*", http://www.dlr.de/IT/IV/Studien/evasoft_abschlussbericht.pdf, BMBF '00

[BS98] Born, Strick: "*Development Tools for Distributed Services*", European Research Consortium for Informatics and Mathematics News No.34, Sophia Antipolis, France '98

[BSH 01] Born, Schieferdecker, Hoffmann: "*The Deployment and Configuration Language*", EURESCOM Workshop on Middleware in Telecommunications, Kjeller, Norway '01

[BSL 00] Born, Schieferdecker, Li: "*Test Framework for Component-Based Systems*", 20th International Conference on Distributed Computing Systems (ICDCS' 2000) with International Workshop on Distributed System Validation and Verification (DSVV'2000), Taipei, Taiwan '00

[BSL 00a] Born, Schieferdecker, Li: "*UML Framework for automated Generation of component based test systems*", 1st International Conference on Software Engineering Applied to Networking & Parallel/ Distributed Computing (SNPD'00), Reims, France '00

[CCMP00] Research Project Description: "*Implementation of CORBA Component Model Session Container*", Berlin, Germany '00

[ComPoTel 00] Research Project Description: "*Component Oriented Design in Telecommunications - Concepts, Notation and Middleware Platform Support*", TINA Fellowship Program, Red Bank, USA '00

[*CoRE* I] Born, Kath: "*CoRE - Komponentenorientierte Entwicklung offener, verteilter Softwaresysteme im Telekommunikationskontext, Band I - Entwicklungsprozesse und Entwicklungstechniken*"

[*CoRE* II] Born: "*CoRE - Komponentenorientierte Entwicklung offener, verteilter Softwaresysteme im Telekommunikationskontext, Band II: Konzeptraum und Notationen*"

[*CoRE* III] Kath: "*CoRE - Komponentenorientierte Entwicklung offener, verteilter Softwaresysteme im Telekommunikationskontext, Band III: Plattformunterstützung und Ableitungsregeln für Softwarekomponenten*"

[DBAG98] "*AEROSPACE*", Magazine of Daimler-Benz Aerospace AG, 1/98, Stuttgart, Germany '98

[DBB+ 01] Dubois, Born, Boehme, Fischer, Holz, Kath, Neubauer, Stoinski: "*Distributed Systems: From Models to Components*", Proceedings of the 10th SDL Forum Conference, Copenhagen, Denmark '01

[DSTCdMOF] Distributed Systems Technology Centre: "*dMOF - An OMG Meta Object Facility Implementation*", http://www.dstc.edu.au/products/CORBA/MOF

[EUP715] EURESCOM Project P715 Deliverable: "*Deliverable 2: Experiments on the EURESCOM Services Platform, Final Report*", Heidelberg, Germany '99

[EUP910] EURESCOM Project P910 Deliverable: "*Deliverable 7: Report on Telecommunication Application Domains*", Heidelberg, Germany '01

[EUP924] EURESCOM Project P924: "*Deployment and configuration support for distributed PNO applications*", http://www.eurescom.de/public/projects/P900-series/p924/P924

[EUP924a] EURESCOM Project P924 Deliverable: "*Deliverable 2: Notation and semantics for deployment and initial configuration*", Heidelberg, Germany '01

[FBH+ 99] Fischbeck, Born, Hoffmann, Winkler, Baudis, Böhme, Fischer: "*SDL enhancements and application for the design of distributed services*": Proceedings of the 9th SDL Forum Conference, Montreal, Canada '99

[FFB 98] Fischer, Fischbeck, Born: "*SDL und ODL im Entwicklungsprozess von Telekommunikationssystemen*": König, Langendörfer (eds.), Formale Beschreibungen für Verteilte Systeme, Shaker Verlag, Aachen, Germany '99

[FFH+ 97] Fischbeck, Fischer, Holz, Kath, v. Löwis, Schröder: "*Improving the Development and Validation of Viewpoint Specifications*", Proceedings of the Conference on Formal Methods for Object Oriented Distributed Systems (FMOODS) '97, Centerburry, UK '97

[FHK+ 98] Fischbeck, Holz, Kath, Vogel: "*Flexible Support of ORB Interoperability*", Proceedings of the Interworking '98 Conference, Ottawa, Canada '98

[FK 98] Frolund, Koistinen: "*QML: A Language for Quality of Service Specification*", White Paper, Hewlett-Packard Laboratories '98

[FK 99] Fischbeck, Kath: "*CORBA Interworking over SS.7*", Proceedings of the Conference on Intelligence in Services in Networks, Barcelona, Spain '99

[FKB 00] Fischer, Kath, Born: "*TINA-ODL and Component Based Design*", Proceedings of the 6th International Conference on Object-Oriented Information Systems 2000, London, UK '00

[FKH+ 98] Fischbeck, Kath, Holz, Geipl, Vogel: "*Operational and Stream Interactions in a B-ISDN based Environment*", Proceedings of the INFORMS 1998 Conference, Boca Raton, USA '98

[FTK+00] Funabashi, Toyouchi, Kanai, Uchihashi, Kobayashi, Hakomori, Yoshida, Strick, Born: "*Development of Open Service Collaborative Platform for Coming ECs by International Joint Efforts*" Procceedings of International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet, Rom, Italy '00

[Gart99] Gartner Executive Report: "*Application Integration: Putting the Pieces of the Puzzle Together*" Gartner Group '99

[Gart99a] Gartner Strategic Analysis Report: "*Middleware Deployment Trends: Survey of Real-World Enterprise Applications*" Gartner Group '99

[GHJ+ 99] Gamma, Helm, Johnson, Vlissides: "*Elements of Reusable Object-Oriented Software*", Addison-Wesley '99

[HKG+ 97] Holz, Kath, Geipl, Lin, Vogel: "*The CAMOUFLAGE Project -Introduction Of TINA Into Telecommunication Legacy Systems*", Proceedings of the TINA 1997 Conference, Santiago de Chile, Chile '97

[HWB 97] Hoffmann, Winkler, Born, Fischer, Fischbeck: *"Towards a Behavioural Description of ODL"* Proceeding of the TINA 1997 Conference, Santiago, Chile '97

[HV 99] Henning, Vinoski: "*Advanced CORBA Programming with C++*", Addison-Wesley '99

[InstallShield] InstallShield Software Corporation: "InstallShield 6.3", http://www.installshield.com/

[ITUTX.292] ITU-T Recommendation X.292: *"OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – The Tree and Tabular Combined Notation (TTCN)"*, ITU '98

[ITUTX.608] ITU-T Recommendation X.608: "*Abstract Syntax Notation One*", ITU-T '99

[ITUTX.902] ITU-T Recommendation X.902 |ISO/IEC 10746-2: *"Open Distributed Processing - Reference Model Part 2"*, ITU-T/ISO '95

[ITUTX.903] ITU-T Recommendation X.903 | ISO/IEC 10746-3: *"Open Distributed Processing - Reference Model Part 3"*, ITU-T/ISO '95

[ITUTX.904] ITU-T Recommendation X.904 | ISO/IEC 10746-4: *"Open Distributed Processing - Reference Model Part 4"*, ITU-T/ISO '95

[ITUTZ.100] ITU-T Recommendation Z.100: "*Specification and Description Language*", ITU-T '00

[ITUTZ.109] ITU-T Recommendation Z.109: "*SDL combined with UML*", ITU-T '00

[ITUTZ.130] ITU-T Recommendation Z.130: "*The ITU-T Object Definition Language*", ITU-T '99

[Kath 99] Kath: "*How to get Behavioural Information on CORBA Systems?*" Proceedings of the Workshop on the Application of Distributed Object Technology, Heidelberg, Germany '99

[KHF+97] Kath, Holz, Fischer, Geipl, Vogel: "*Provision of TINA Object Interaction through B-ISDN in ATM networks*", Proceedings of the GLOBECOM Conference, Phoenix, USA '97

[KKS00] Kath, Kleinschmidt, Stoinski: *"CPE Architecture for TINA Services (CATS II): Final Project Deliverable"*, document number CATS-II HU 006, Tokio, Japan '00

[KS00] Kath, Stoinski: "*Project Proposal: Platform for Authoring and Composition of Interactive Content (PACIFIC)*", Projektvorbereitungsdokument, Berlin, Germany '00

[KST+ 01] Kath, Stoinski, Takita, Tsuchiya: "*Middleware Platform Support for Multimedia Content Provision*", Proceedings of the 3G Technologies and Applications Conference, Heidelberg, Germany '01

[KT 99] Kath, Takita: "*OMG A/V Streams and TINA NRA: An integrative Approach*", Proceedings of the TINA 1999 Conference, Oahu, USA '99

[KvHS+ 00] Kath, v. Halteren, Stoinski, Wegdam, Fisher: "*Middleware Platform Management Based on Portable Interceptors*", Proceedings of 11th Annual IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Austin, USA '00

[MDTS 01] Forschungs- und Entwicklungsprojekt: "*Model Driven Development of Distributed Telecommunication Systems*", Bundesministerium für Bildung und Forschung, Berlin, Germany '01

[MICO] Römer, Puder et. al.: "*MICO for C++*", http://www.mico.org/

[MS .net] Microsoft Corporation: "*The .NET Framework*", Micosoft Developer Network Online, http://msdn.microsoft.com/net/framework/default.asp

[MS C#] Microsoft Corporation: "*C# Introduction and Overview*", Micosoft Developer Network Online, http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp

[MS COM] Microsoft Corporation: "*The Component Object Model*", Microsoft Press '98

[Neu 95] Neumann: "*Objektorientierte Entwicklung von Software-Systemen*", Addison-Wesley '95

[OMGASRFP] Object Management Group: *"Action Semantics for the UML Request For Proposal"* OMG document ad/98-11-01

[OMGAVStreams] Object Management Group: "*Control and Management of Audio/Video Streams*", OMG document formal/00-01-03

[OMGCCM I] BEA Systems et. al.: *"CORBA Components - Volume I"*, OMG document orbos/99-07-01

[OMGCCM II] BEA Systems et. al.: "*CORBA Components - Volume II*", OMG document orbos/99-07-02

[OMGCCM III] BEA Systems et. al.: "*CORBA Components - Volume III*", OMG document orbos/99-07-03

[OMGCCM RFP] Object Management Group: "*CORBA Component Model RFP, Final Version*", OMG document orbos/97-06-12

[OMGCORBA] Object Management Group: „*The Common Object Request Broker: Architecture and Specification, Revision 2.4.2*", OMG document formal/01-02-33

[OMGCORBAES] Object Management Group: "*Event Service Specification, Version 1.1*", OMG document formal/01-03-01

[OMGCORBAIDL] Object Management Group: „CORBA 2.4.2 IDL Syntax and Semantics", OMG document formal/01-02-39

[OMGCORBANaS] Object Management Group: "*CORBA Naming Service*", OMG document formal/01-02-65

[OMGCORBANoS] Object Management Group: "*Notification Service Specification, Version 1.0*", OMG document formal/00-06-20

[OMGCORBAP] Data Access Corporation et. al.: "*UML Profile for CORBA*", OMG document ad/00-02-02

[OMGCWM] Object Management Group: "*Common Warehouse Metamodel (CWM) Specification, Version 1.0*", OMG document ad/01-02-01

[OMGC++Map] Object Management Group: "*C++ Language Mapping Specification*", OMG document formal/99-07-41

[OMGDeplRFP] Object Management Group: "*Deployment and Configuration of Distributed Applications Draft RFP*", OMG document orbos/01-04-12

[OMGMDA] Object Management Group: "*Model Driven Architecture*", OMG document omg/00-11-05

[OMGMICRFP] Object Management Group: "*Multiple Interfaces and Composition RFP*", OMG document orb/96-01-04

[OMGMOF1.3] Object Management Group: "*Meta Object Facility, Version 1.3*", OMG document formal/00-04-03

[OMGMOFP] Object Management Group: "*UML Profile for MOF*", OMG document ad/01-02-29

[OMGPI] BEA Systems et. al.: "*Portable Interceptors*", OMG document orbos/99-12-02

[OMGPSS2.0] Object Management Group: "*Persistent State Service 2.0*", OMG document orbos/99-07-07

[OMGReqUMLP] Object Management Group: "*Requirements for UML Profiles*", OMG document ad/99-12-32

[OMGSPEM 01] IBM, Rational Software, Fujitsu/DMR, SOFTEAM, Unisys, Nihon Unisys Ltd., Alcatel, Q-Labs: "*Software Process Engineering Management: The Software Process Engineering Metamodel (SPEM)*", OMG document ad/2001-03-08

[OMGSPERFP] Object Management Group: "*Software Process Engineering (SPE) Management Request for Proposal*", OMG document ad/99-11-04

[OMGUML1.3] Object Management Group: "*OMG Unified Modeling Language Specification, Version 1.3*", OMG document ad/99-06-08

[OMGXMI1.1] Object Management Group: "*XML Metadata Interchange (XMI) version 1.1*" OMG document formal/00-11-02

[OOC] Object Oriented Concepts Inc.: "*ORBacus™ for C++ and Java*", http://www.ooc.com/ob/

[OOCa] Object Oriented Concepts: "*Customer Success Stories*", http://www.ooc.com/customers/telecom.html

[OOCJTC] Object Oriented Concepts Inc.: "*Java™-like Threads for C++*", http://www.ooc.com/jtc/

[Pom 91] Pomberger: "*Prototyping-Oriented Software Development - Concepts and Tools*", Structured Programming, Vol. 12, Nr. 1 '91

[PS 94] Pagel, Six: "*Software Engineering - Band 1: Die Phasen der Softwareentwicklung*", Addison-Wesley '94

[Rational] Rational Software Corp.: "*Customer Stories*", http://programs.rational.com/success/Success_Result.cfm

[RationalRose] Rational Software Corp.: "*Rational Rose*", http://www.rational.com/products/rose/index.jsp/

[RationalRUP] Rational Software Corp.: "*Rational Unified Process: Best Practices for Software Development Teams*" Rational Software Corp. White Paper, http://www.rational.com/products/whitepapers/100420.jsp

[Ray95] Raymond: "*Reference Model of Open Distributed Processing (RM-ODP): Introduction*" Proceedings of International Conference of Open Distributed Processing, Brisbane, Australia '95

[RFCMIME] Internet RFC 2046: "*Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*", IETF '98

[Roy 70] Royce: "*Managing the Development of Large Scale Software Systems*", Proceedings of WESCON Conference, Los Angeles, USA '70

[SB97] Strick, Born: "*Developing TINA Services with ODL and SDL*" Proceedings of the Workshop on Distributed Object Technologies for Telecoms Networks (DOT '97), Heidelberg, Germany '97

[Sun 01] Sun Microsystems White Paper: „*Network Application Strategies for Telecom: Introduction Component Architectures for Enhanced Services*", http://www.sun.com/software/solutions/third-party/software/whitepapers/Java.Telco.pdf

[Sun EJB] Sun Microsystems: "*Enterprise JavaBeans TM*" http://www.sun.com

[Sun EJBa] Sun Microsystems: "*Industry Opinions On Enterprise JavaBeansTM (EJBTM) vs. COM+/MTS*", http://java.sun.com/products/ejb/ejbvscom.html

[Sun JAVA] Sun Microsystems: "*Java 2 Platform Standard Edition*", http://www.javasoft.com/j2s

[Sun SSL] Sun Microsystems, Netscape Corp.: "*Introduction to SSL*", http://docs.iplanet.com/docs/manuals/security/sslin/index.htm

[Szy99] Szyperski "*Component Software - Beyond Object-Oriented Programming*", Addison-Wesley '99

[TINA] TINA-C: "http://www.tinac.com"

[TINACMC] TINA-C: "*Computational Modeling Concepts*",TINA-C '98

[TINAReq] TINA-C: "*Requirements upon TINA-C Architecture*", TINA-C '95

[TINASA] TINA-C: "*TINA Service Architecture 5.1*", TINA-C '97

[TTK+ 00] Tsuchiya, Takita, Kath, Stoinski: "*Authoring, Composition, and Delivery of Interactive Contents by the use of Multimedia Contents Mill*", Proceedings of the TINA 2000 Conference, Paris, France '00

[TTK+ 00a] Tsuchiya, Takita, Kath, Stoinski: "*'Multimedia Contents Mill' – A Platform for Authoring and Delivery of Interactive Multimedia Contents*", Proceedings of the SoftCom 2000 Conference, Split, Croatia '00

[UnisysUREP] Unisys Corp.: "*Universal Repository*", http://www.unisys.com/marketplace/urep/

[vHalt00] A.T. van Halteren: "*A reflective QoS provisioning service for object middleware*", Workshop on Reflective Middleware (RM 2000), co-located with the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '00), New York, USA '00.

[VSB+ 99] Vassiliou-Gioles, Schieferdecker, Born, Winkler, Li: "*Configuration and Execution Support for Distributed Tests*" Proceedings of the 12th International Workshop on Testing of Communicating System (IWTCS 99), Budapest, Hungary '99

[Web01] Weber: "*Quo Vadis Software Engineering*", Gesellschaft für Informatik, Softwaretechnik-Trends, Band 21 Heft 1, http://pi.informatik.uni-siegen.de/stt/21_1/index.html, GI '00

[WS 96] Weck, Szyperski: „*Do We Need Inheritance?*" Proceedings of the Workshop on Composability Issues in Object-Orientation (at ECOOP'96), Linz, Austria '96

[WWF+ 95] Wasowski, Witaszek, Fischer, Holz, Lau, Kath: "*Co-Simulation of Hybrid SDL and VHDL Specifications*", Proceedings of the 9th ESM Conference, Prague, Chzech Republic '95

[W3CHTTPNG] Janssen et. al.: „*HTTP-ng Architectural Model*", W3C Internet Draft, www.w3c.org

[W3COSD] Marimba Incorporated, Microsoft Corporation: "*The Open Software Description Format (OSD)*", submitted to W3C, http://www.w3.org/TR/NOTE-OSD.html, W3C '97

[W3CSMIL] W3C Recommendation: "*Synchronized Multimedia Integration Language (SMIL 2.0) Specification* ", W3C Proposed Recommendation, http://www.w3.org/TR/2001/PR-smil20-20010605/, W3C '01

[W3CXML] W3C Recommendation: "*Extensible Markup Language (XML) 1.0 (Second Edition)*", http://www.w3.org/TR/2000/REC-xml-20001006, W3C '00

[W3CXMLDT] W3C Recommendation: "*XML Schema Part 2: Datatypes*", http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/, W3C '01

ABKÜRZUNGEN

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CCM	CORBA Component Model
CDR	Common Data Representation
CIDL	Component Implementation Definition Language
CIF	Component Implementation Framework
CO	Computational Object
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CoRE	Components Rapid Engineering
DCOM	Distributed COM
DTD	Document Type Definition
EJB	Enterprise Java Beans
HTTP-NG	HTTP Next Generation
IDL	Interface Definition Language

IIOP	Internet Inter-ORB Protocol
OCL	Object Constraint Language
ODL	Object Definition Language
OMG	Object Management Group
OSD	Open Software Description
MDA	Model Driven Architecture
MOF	Meta Object Facility
MP3	MPEG Audio Layer-3
MPEG	Moving Picture Experts Group
RFP	Request for Proposals
RMI	Remote Methode Invocation
RM-ODP	Reference Model for Open Distributed Processing
RUP	Rational Unified Process
SDL	Specification and Description Language
SPEM	Software Process Engineering Metamodel
TINA	Telecommunications Information Networking Architecture
UDP	User Datagram Protocol
UML	Unified Modeling Language
UUID	Universal Unique Identifier
XMI	XML Metadata Interchange
XML	Extensible Markup Language

INDEX

ANHANG A *Eingeführte Termini*

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Artefakt	*artifact*	Abstraktion konkreter programmiersprachlicher, in Softwarekomponenten in Form von Codemodulen enthaltener Konstrukte (z.B. Klassen im Falle von objektorientierten Programmiersprachen) Artefaktinstanzen realisieren das Verhalten, den Zustand und die Identität von COs	CCM
Assemblage	*assembly*	Beschreibung eines verteilten Softwaresystems durch Angabe der beteiligten CO-Typen und deren initialer Konfiguration	CCM
Attribut (eines Interfacetyps)	*attribute*	Spezielle Variante von Operationen im Sinne einer verkürzten Schreibweise für *get*- und *set*-Operationen für einen bestimmten Datentyp	CORBA
Ausführungszeit	*runtime*	Zeit, in der die in Softwarekomponenten enthaltenen Codemodule durch eine Maschine ausgeführt werden	
Ausnahme	*exception*	Spezielle Art der Operationsterminierung im Falle von Fehlern	RM-ODP

Tab.8 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Bindung	*binding*	Konzept zur Beschreibung des Austausches von Referenzen auf Instanzen der zu *Port*-Definitionen gehöhrenden Interfacetypen unter Beachtung der Art der *Port*-Definition (*Used*- oder *Provided- Port*-Definition)	RM-ODP, CCM
Bindungsfall	*binding case*	Zusammenfassung von COs, Bindungen dieser COs und zu diesen Bindungen gehörigen Bindungsregeln	
Bindungsregel	*binding rule*	Beschreibung von geforderten Eigenschaften von Interaktionen, die auf der Grundlage von Bindungen entstehenden, formuliert als Regeln über Kontrakttypen	RM-ODP
Component-Support- Plattform	*component support platform*	Plattform, die *Deployment*- und Ausführungsunterstützung für Softwarekomponenten unter Benutzung einer *Distributed-Processing*-Umgebung bereitstellt	
Computational- Objekt	*computational object*	Objekt, das durch funktionale Dekomposition eines Softwaresystems während dessen Modellierung entsteht	RM-ODP, TINA
Computational- Objekttyp	*computational object type*	Beschreibung von *Computational*-Objekten, die zu deren Instanziierung benutzt wird	RM-ODP, TINA
Consume-Definition	*consume*	Interaktionselement der Signalinteraktion, beschreibt durch Angabe eines Signaltyps die Möglichkeit des Konsumierens eines Signals dieses Signaltyps im Kontext eines Interfaces	CCM
Continuous-Media-, Signal- und operationale Interaktion	*continous media, signal and operational Interaction*	Interaktion zwischen COs unter Verwendung von operationalen, Signal- oder *Continous-Media*-Interaktionselementen (Operation, Attribut, *Consume*, *Produce*, *Sink*, *Source*)	RM-ODP, TINA (operationale und *Continous-Media*-Interaktion)
Datentyp	*data type*	Element eines Datentypsystems (z.B. CORBA-IDL) und Basis von Informationsmodellen, dient der strukturierten Repräsentation von Information	CORBA
Distributed-Processing- Umgebung	*distributed processing environment*	Technologische Grundlage zur Unterstützung der Interaktion von Objekten eines verteilten Softwaresystems	TINA
Implementierungs-element	*implementation element*	Beschreibung einer Relation zwischen Interaktionselement und Artefakt mit der Bedeutung, daß eine Instanz des Artefakts für die Realisierung des Verhaltens des Interaktionselements verantwortlich ist.	
Implements-Relation	*implements*	Relation zwischen Artefakten und CO-Typen mit der Bedeutung, daß Instanzen der Artefakte das Verhalten der COs dieses CO-Typs realisieren	

Tab.8 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
initiales CO	*initial CO*	Instanz eines CO-Typs, die zu Beginn der Ausführungszeit des Softwaresystems erzeugt wird	CCM (nicht als CO sondern als CORBA *component* bezeichnet)
initiale Konfiguration	*initial configuration*	Beschreibung der initialen COs und deren initialen Bindungen	CCM
initiale Bindung	*initial binding*	Bindung, die zu Beginn der Ausführungszeit eines Softwaresystems erzeugt wird	CCM
Instanziierungs-muster	*instantiation policy*	Beschreibung, nach welchen Regeln zur Ausführungszeit Instanzen des durch ein Artefakt modellierten programmiersprachlichen Konstruktes angelegt werden sollen	
Interaktionselement	*interaction element*	Generalisierung der Konzepte Operation, Attribut, *Sink*, *Source*, *Consume*, *Produce*	
Interface	*interface*	auf der Grundlage eines Interfacetyps im Kontext eines COs entstehende referenzierbare Zusammenfassung von potentiellen Interaktionen eines COs	RM-ODP
Interfacetyp	*interface type*	Aggregation von Interaktionselementen als benannter, identifizierbarer Endpunkt von potentieller Interaktion	RM-ODP
Kontrakttyp	*contract type*	Datentyp, der Elemente zur Definition von Eigenschaften der aufgrund von Bindungen zustandekommenden Interaktionen beinhaltet	QML
Konzeptraum	*concept space*	Die Gesamtheit aller für die objektorientierte Modellierung in einem Anwendungsgebiet verwendbaren Konzepte und deren Beziehungen	
Maschine	*node*	Gerät, das zur Ausführung der in Softwarekomponenten enthaltenen Codemodule geeignet ist (i.allg. Computer)	RM-ODP
Medienmenge	*media set*	Aggregation von Medien	
Medientyp	*media type*	Vorschrift für die Codierung, die Übertragung und Decodierung der Mediendaten eines Mediums	
Medium	*medium*	Abstraktion einer multimedialen Information	
Metamodell	*metamodel*	Modell, daß das Instrumentarium zur Definition von Modellen beschreibt	MOF
Modellklasse	*viewpoint*	Definition einer Einschränkung auf bestimmte Aspekte von Entwurfsmodellen durch Angabe von hierfür relevanten Konzepten und Beziehungen eines Konzeptraumes	RM-ODP

Tab.8 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Multiple-Port-Definition	*multiple port*	*Port*-Definition, auf deren Grundlage eine Vielzahl von Interfacereferenzen zur Ausführungszeit eines COs hinterlegbar oder beschaffbar ist	CCM (dort nur für benutzte Interfacetypen)
Namensraum	*namespace*	Konzept zur Strukturierung von Namen der Elemente eines Modells	RM-ODP
Objekt	*object*	Modell einer Entität von Interesse in der betrachteten Anwendungsdomäne	RM-ODP
Objektumgebung	*environment of an object*	Teil des Softwaresystems, der nicht Bestandteil des Objektes ist	RM-ODP
Operation	*operation*	Interaktionselement der operationalen Interaktion, beschrieben durch Parameter und mögliche Arten der Terminierung	RM-ODP, CORBA
Parameter	*parameter*	Identifizierbarer Bestandteil einer Operation, definiert die Richtung des Informationsflusses bei der Interaktion und einen der Information zugrundeliegenden Datentyp	CORBA
Port-Definition	*port*	Beschreibung der Möglichkeit der Hinterlegung bzw. Beschaffung von Referenzen auf Interfaces eines entsprechenden COs zur Ausführungszeit	CCM
Prädikat	*predicate*	Abstraktion von zu wahr oder falsch evaluierbaren Ausdrücken	RM-ODP
Produce-Definition	*produce*	Interaktionselement der Signalinteraktion, beschreibt durch Angabe eines Signaltyps die Möglichkeit des Versendens eines entsprechenden Signals im Kontext eines Interfaces	CCM
Provided-Port-Definition	*provides*	auf einer *supports*-Relation basierende *Port*-Definition zur Beschreibung der Möglichkeit der Beschaffung von Referenzen auf Interfaces eines COs zur Ausführungszeit	CCM
Realize-Relation	*realize*	Zuordnung von CO-Typen zu Softwarekomponenten	UML
Requires-Relation	*requires*	Relation zwischen Interfacetyp und CO-Typ mit der Bedeutung, daß COs dieses CO-Typs Interfaces dieses Interfacetyps von ihrer Umgebung erwarten	TINA
Sicht	*viewpoint*	s. Modellklasse	RM-ODP
Signal	*signal*	Atomare Nachricht, die asynchron und entkoppelt zwischen COs ausgetauscht wird, entsteht auf der Basis eines Signaltyps	RM-ODP

Tab.8 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Signalparameter	*signal parameter*	Identifizierbare Angabe eines Datentyps im Kontext eines Signaltyps	RM-ODP
Signaltyp	*signaltype*	Beschreibung von Signalen, die zu deren Instanziierung im Kontext von Signalinteraktion verwendet wird, Aggregation von Signalparametern	RM-ODP
Single-Port-Definition	*single port*	*Port*-Definition, auf deren Grundlage eine einzelne Interfacereferenz zur Ausführungszeit eines COs hinterlegbar oder beschaffbar ist	CCM
Sink-Definition	*sink*	Interaktionselement der *Continous-Media*-Interaktion, beschreibt durch Angabe einer Medienmenge die Möglichkeit des Empfangs der Medien auf der Basis eines geeigneten Medientyps im Kontext eines Interfaces	TINA
Software-komponente (modelliert)	*software component*	Abstraktion einer Softwarekomponente im Modell (Abstraktion von den Instruktionssequenzen)	[Szy99]
Software-komponente (real)	*software component*	physikalisch repräsentierte Entität, bestehend aus in Codemodulen zusammengefaßten Instruktionssequenzen, die Ausführung einer Softwarekomponente führt zur Inkarnation von Objekten	[Szy99]
Softwarepaket	*software package*	Zusammenfassung von Softwarekomponenten eines realen Softwaresystems	CCM, OSD
Softwaresystem (real)	*software system*	System bestehend aus Softwarekomponenten	[Szy99]
Source-Definition	*source*	Interaktionselement der *Continous-Media*-Interaktion, beschreibt durch Angabe einer Medienmenge die Möglichkeit des Sendens der Medien auf der Basis eines geeigneten Medientyps im Kontext eines Interfaces	TINA
supports-Relation	*supports*	Relation zwischen Interfacetyp und CO-Typ mit der Bedeutung, daß COs dieses CO-Typs Interfaces dieses Interfacetyps ihrer Umgebung bereitstellt	TINA
Terminierung	*termination*	Ende einer gerufenen Operation	RM-ODP
Used-Port-Definition	*uses*	auf einer *requires*-Relation basierende *Port*-Definition	CCM
Zustandsattribut	*state attribute*	spezieller Datentyp zur Repräsentation von Zustandsinformationen von CO-Typen	RM-ODP, CCM

Tab.8 Verwendete Termini in der Übersicht

ANHANG B Fallbeispiel

In diesem Anhang werden die Entwicklungstechniken von *CORE* zur Entwicklung eines verteilten Softwaresystems eingesetzt. Damit wird die Anwendung der Konzepte von $CORE_{CEPT}$ zur Modellbildung, die Darstellung des Entwurfsmodells mit den Notationen von $CORE_{TATIONS}$ sowie die Anwendung von $CORE_{MAP}$ zur automatischen Ableitung von Softwarekomponenten aus Modellen demonstriert. Die für diese Entwicklungsaufgabe verwendete Komponentenarchitektur ist $CORE_{WARE}$. Als Entwicklungsprozeß wird hier aufgrund der eingeschränkten Komplexität des Beispiels das klassische Phasenmodell ohne Iterationen verwendet.

B.1 Analyse und Definition

Das zu entwickelnde Softwaresystem soll das klassische Beispiel „*Dining Philosophers*" implementieren. Dieses Anwendungsbeispiel wurde von Edger W. Dijkstra 1965 erstmalig formuliert. Es ist ein Modell und eine universale Methode zum Testen und Vergleichen von Theorien über das Alloziieren von Ressourcen. Das Problem besteht aus einer endlichen Menge von Prozessen die eine ebenfalls endliche Menge von Ressourcen teilen. Die Ressourcen können immer nur von einem Prozeß gleichzeitig verwendet werden - damit können sowohl *Lifelock*- als auch *Deadlock*- Situationen entstehen.

Eine Konkretisierung des Problems kann wie folgt beschrieben werden: Eine Menge von Philosophen versammelt sich um einen Tisch, auf dem eine Menge von Gabeln bereitsteht. Philosophen führen die Aktionen Essen, Schlafen und Denken aus. Zum Essen benötigen sie zwei Gabeln - eine für die rechte Hand und eine weitere für die linke Hand. Die Gabeln werden den Philosophen aus der Menge der verfügbaren Gabeln zugewiesen, d.h. ein Philosoph muß zum essen immer die ihm zugeordneten Gabeln verwenden. Gabeln können nur von jeweils einem Philosophen zu einem Zeitpunkt benutzt werden.

Zur Visualisierung des Verhaltens von Philosophen gibt es einen Observierer (*Observer*). Dieser soll von den Philosophen über die Veränderung ihrer aktuell ausgeführten Tätigkeit informiert werden. Diese Situation ist in Abb.83 verdeutlicht.

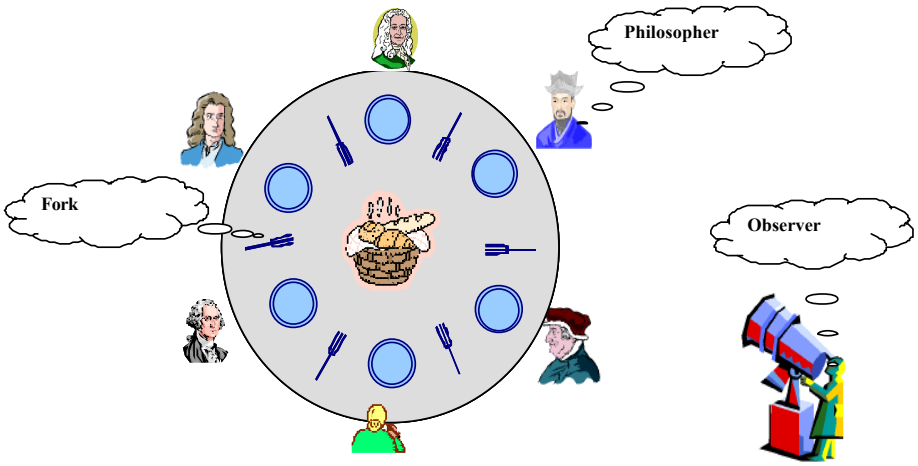


Abb. 83 Das Beispiel „Dining Philosphers“

Das beschriebene Beispiel ist so zu implementieren, daß Philosophen, Observierer und Gabeln auf unterschiedlichen Maschinen ausgeführt werden und trotzdem miteinander interagieren können. Es werden keine zusätzlichen nicht-funktionalen Anforderungen an das Softwaresystem gestellt.

B.2 Entwurf

STRUKTURSICHT. In der Struktursicht sind die Grundelmente für die Interaktionen zwischen den COs eines verteilten Softwaresystems zu spezifizieren.

Für das hier betrachtete Beispiel ist konkret eine Signal-basierte Interaktion zwischen Philosophen und dem Observierer zu modellieren. Hierzu wird ein Signaltyp ***PhilosopherState*** definiert, der als Parameter den Datentyp ***PState*** unter dem Namen ***carry_pstate*** besitzt. Elemente dieses Datentyps sind der Name des Philo-

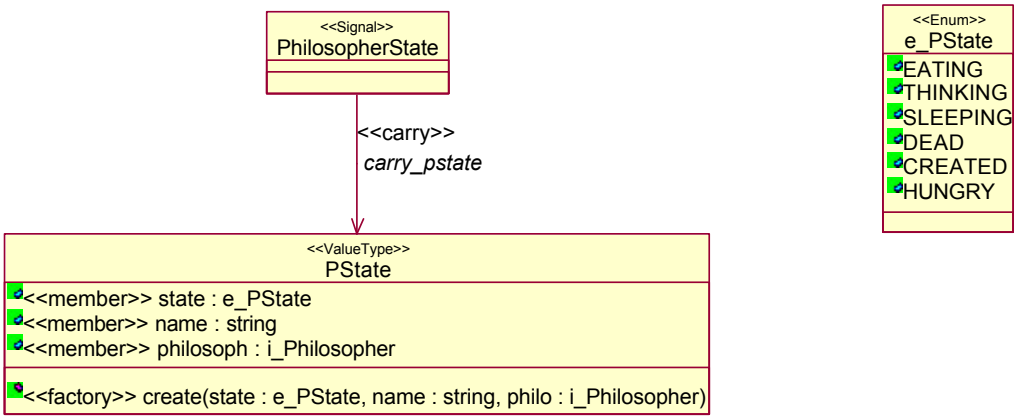


Abb. 84 Signal- und Signalparameter

sophen, der Zustand des Philosophen sowie ein weiteres Element, das als Typ einen Interfacetyp besitzt, dessen Bedeutung später beschrieben wird. Der Zustand eines Philosophen wird durch den Datentyp ***e_Pstate*** beschrieben. Alle diese Modellelemente sind unter Verwendung von *CoRE*$_{TATIONS}$ in Abb.84 notiert.

Auf der Basis der definierten Elemente können nun die Interfacetypen zur Interaktion zwischen COs eingeführt werden (vgl. Abb. 85):

- Der Interfacetyp ***i_Observer*** beinhaltet das *Consume*-Interaktionselement ***pstate*** zum Empfang von ***PhilosopherState***,
- der Interfacetyp ***i_Philosopher*** beinhaltet die Operation ***set_name*** mit dem Parameter ***name*** des Datentyps ***string***,
- der Interfacetyp ***i_Fork*** beinhaltet die Operationen ***obtain_fork*** und ***release_fork***; jeweils mit dem Parameter ***requester*** vom Typ ***o_Philosopher*** zur Identifikation des Philosophen, der eine Gabel benutzen oder freigeben möchte. Der Operation ***obtain_fork*** ist die Ausnahme ***ForNotAvailable*** zugeordnet, die ausgelöst wird, falls eine Gabel zum Zeitpunkt des Operationsrufes nicht verfügbar ist, d.h. von einem anderen Philosophen benutzt wird. Der Operation ***release_fork*** ist die Ausnahme ***NotTheEater*** zugeordnet, die ausgelöst wird, falls der Rufer dieser Operation nicht der aktuelle Besitzer der Gabel ist.

Für dieses Beispiel werden Philosophen, Gabeln und Observierer durch die CO-Typen ***o_Philosopher***, ***o_Observer*** und ***o_Fork*** modelliert.

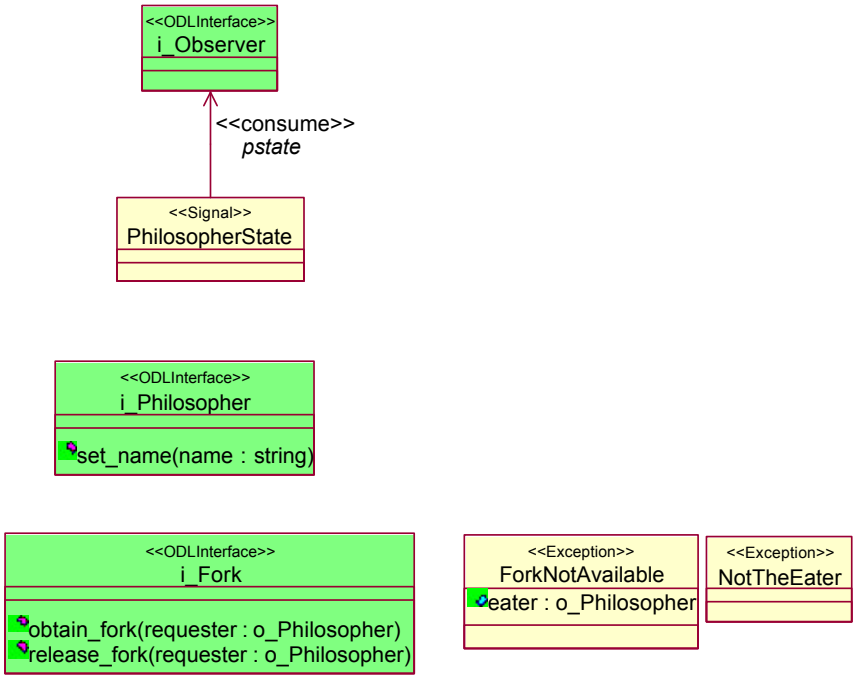


Abb. 85 Interfacetypen

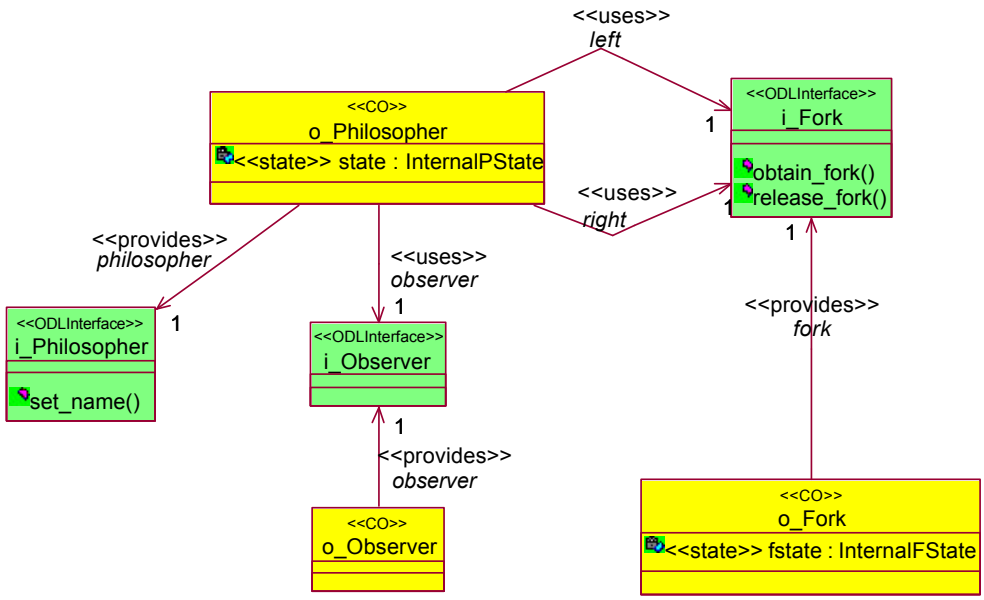


Abb. 86 CO-Typen

KONFIGURATIONSSICHT. Für die definierten CO-Typen werden folgende *Single-Port*-Definitionen vorgenommen (vgl. Abb. 86):

- ***o_Fork*** enthält eine *Provided-Port*-Definition mit dem Namen ***fork***, an der der Interfacetyp ***i_Fork*** angeboten wird,
- ***o_Observer*** enthält eine *Provided-Port*-Definition mit dem Namen ***observer***, an der der Interfacetyp ***i_Observer*** angeboten wird,
- ***o_Philosopher*** enthält eine *Provided-Port*-Definition mit dem Namen ***philosopher***, an der der Interfacetyp ***i_Philosopher*** angeboten wird,
- ***o_Philosopher*** enthält zwei *Used-Port*-Definitionen mit den Namen ***left*** und ***right***, an denen der Interfacetyp ***i_Fork*** genutzt wird,
- ***o_Philosopher*** enthält eine *Used-Port*-Definition mit dem Namen ***observer***, an der der Interfacetyp ***i_Observer*** genutzt wird.

IMPLEMENTIERUNGSSICHT. Die CO-Typen werden durch Artefakte wie folgt realisiert (vgl. Abb. 87):

- ***o_Fork*** wird durch das Artefakt ***o_Fork_Impl*** mit den Implementierungselementen ***obtain_fork_impl*** und ***release_fork_impl*** für die Interaktionselemente ***obtain_fork*** und ***release_fork*** realisiert,
- ***o_Observer*** wird durch das Artefakt ***o_Observer_Impl*** mit dem Implementierungselement ***pstate_in*** für das Interaktionselement ***pstate*** von ***i_Observer*** realisiert,
- ***o_Philosopher*** wird durch das Artefakt ***o_Philosopher_Impl*** mit den Implementierungselementen ***set_name_impl*** für das Interaktionselement ***set_name*** von ***i_Philosopher*** und ***pstate_out*** für das Interaktionselement ***pstate*** von ***i_Observer*** realisiert.

Der CO-Typ ***o_Philosopher*** erhält ein Zustandsattribut ***state*** vom Typ ***InternalPState***, der CO-Typ ***o_Fork*** ein Zustandsattribut ***fstate*** vom Typ ***InternalFState***.

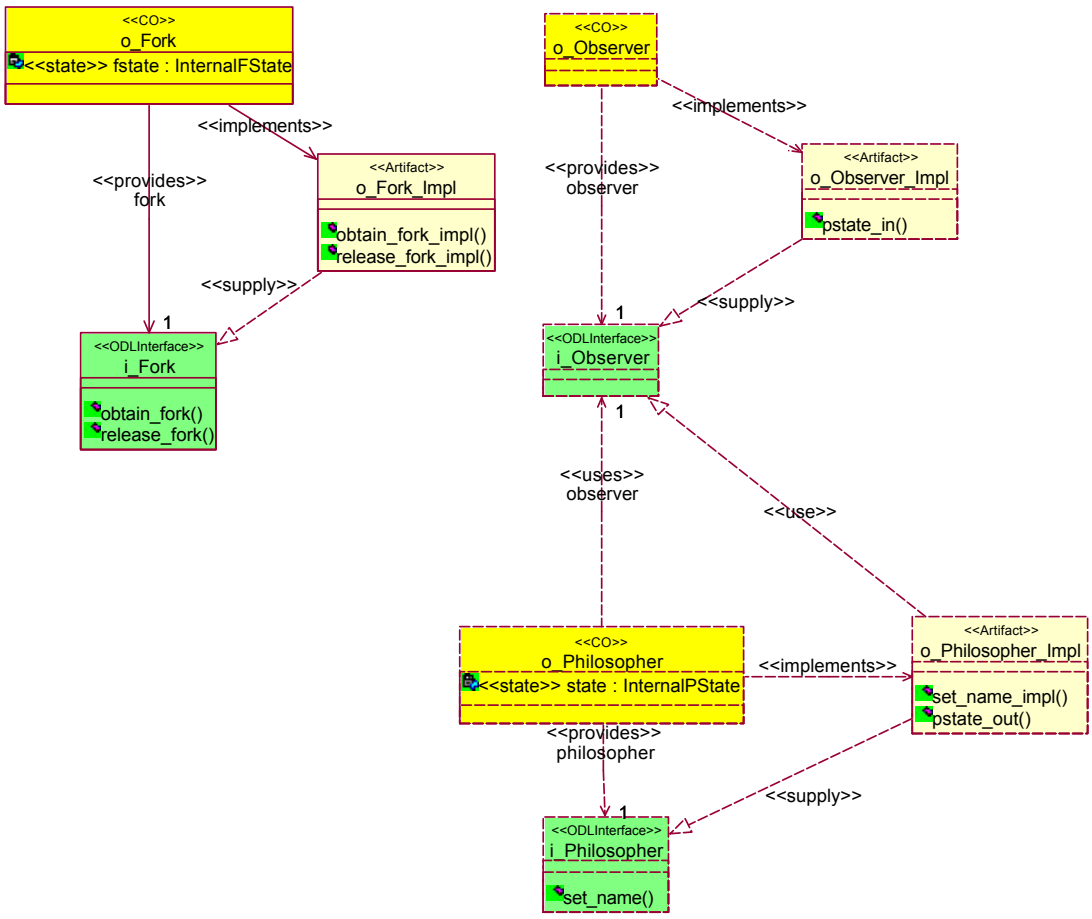


Abb. 87 Artefakte

DEPLOYMENT-SICHT. Die CO-Typen werden Softwarekomponenten wie folgt realisiert (vgl. Abb. 88)

- ***o_Philosopher*** und ***o_Fork*** werden durch die Softwarekomponente ***Philosopher*** bereitgestellt,
- ***o_Observer*** wird durch die Softwarekomponente ***Observer*** realisiert.:

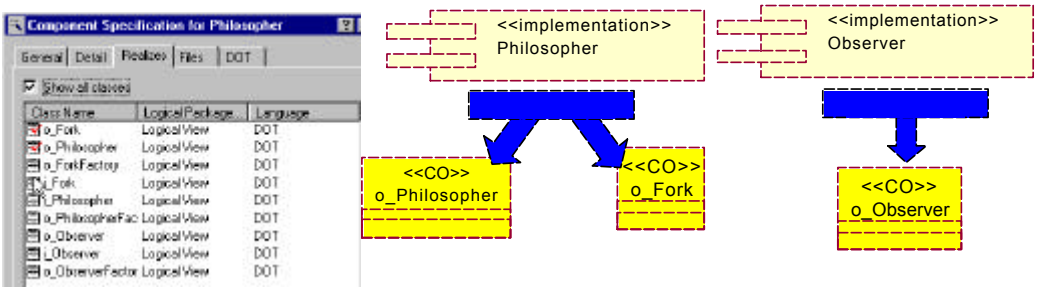


Abb. 88 Softwarekomponenten

INTERAKTIONSSICHT. Da keine spezifischen nicht-funktionalen Anforderungen an das zu entwickelnde Softwaresystem in der Analyse- und Definitionsphase gestellt wurden, sind keine diesbezüglichen Entwurfsdefinitionen notwendig.

B.3 Implementierung

Durch *CORE*$_{MAP}$ werden CORBA-IDL-Interfacedefinitionen und C++-Codemodule erzeugt. Die Codemodule werden vom Entwickler komplettiert. Folgende Implementierungsaktivitäten sind dabei auszuführen:

- Implementierung der Artefaktklassen für ***o_Observer_Imp***, ***o_Fork_Impl*** und ***o_Philosopher_Impl*** sowie
- die Vervollständigung der main-Methode der entsprechenden CO-Klassen.

Der durch den Entwickler definierte C++-Code ist im folgenden dargestellt.

```
void o_Fork_Impl::obtain_fork_impl
( ::Philosophers::o_Philosopher_ptr requester, const char * name )
 {
  JTCSynchronized sync ( * dynamic_cast < CO_o_Fork * > (co())->state()->fstate() );
  if ( dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->state == USED )
    throw ForkNotAvailable ();
  dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->state = USED;
  dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->eater
    = ::Philosophers::o_Philosopher::_duplicate ( requester );
  unsigned long my_number
    = dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->number;
  GuiControl::instance()->use_fork(my_number, name );
 }

 void o_Fork_Impl::release_fork_impl
 ( ::Philosophers::o_Philosopher_ptr requester )
 {
  JTCSynchronized sync ( *dynamic_cast < CO_o_Fork * > (co())->state()->fstate() );
  if ( dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->state == USED )
  {
    dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->state = UNUSED;
    dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->eater
      = ::Philosophers::o_Philosopher::_nil();
    unsigned long my_number
      = dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->number;
    GuiControl::instance()->unuse_fork(my_number);
    Sleep ( 250 );
  }

 void o_Philosopher_Impl::set_name_impl ( const char * name )
 {
  JTCSynchronized sync ( * dynamic_cast<CO_o_Philosopher *> (co())->state()->state() );
  dynamic_cast < CO_o_Philosopher * > (co())->state()->state()->name
    = CORBA::string_dup ( name );
 }

 void CO_o_Philosopher::main ()
 {
  GuiControl::instance()->inc_philosopher();
  srand( (unsigned)time( NULL ) );
  ::Container::Container::instance()->message ( "CO_o_Philosopher::main()" );
  {
    JTCSynchronized sync ( *state()->state() );
    state()->state()->name = CORBA::string_dup ( "PHILOSOPH" );
  }

  {
    JTCSynchronized sync ( *this );
```

```
  while ( CORBA::is_nil ( get_left() ) || CORBA::is_nil ( get_right () ) )
  {
   this->wait();
  }
}

try
{
  ComponentModel::Parameters params;
  this->resolve_port_supply ( "::Philosophers::o_Observer", "observer", params );
  CORBA::Object_var provided_
    = this->resolve_port ( "::Philosophers::o_Observer", "observer" );
  this->co()->connect ( "observer", provided_, params );
  }
catch ( ... )
{
  MESSAGE ( "cannot connect to observer" );
}

send_signal ( this, CREATED );
while ( 1 )
{
  send_signal ( this, HUNGRY );

  bool have_left_fork = false;
  bool have_right_fork = false;
  while ( ! have_left_fork || ! have_right_fork )
  {
    if (!have_left_fork)
    {
      try
      {
        get_left()->obtain_fork ( co(), CORBA::string_dup ( state()->state()->name) );
        have_left_fork = true;
      }
      catch ( ForkNotAvailable& )
      {
        if ( have_right_fork )
        {
          MESSAGE ( "release right fork" );
          get_right()->release_fork ( co() );
          have_right_fork = false;
        }
          Sleep ( rand() % 100 );
      }
    }

    if (!have_right_fork)
    {
      try
      {
        get_right()->obtain_fork ( co(), CORBA::string_dup ( state()->state()->name) );
        have_right_fork = true;
      }
      catch ( ForkNotAvailable& )
      {
        if ( have_left_fork )
        {
          get_left()->release_fork ( co() );
          have_left_fork = false;
        }
          Sleep ( rand() % 100 );
      }
    }
  }
    if ( observer_ )
```

```
      send_signal ( this, EATING );
    Sleep ( rand() % 2000 );
    try
    {
      get_left() -> release_fork ( co() );
      get_right() -> release_fork ( co() );
    }
    catch ( NotTheEater& )
    {
      MESSAGE ( "someone has stolen the fork" );
    }
    send_signal ( this, THINKING );
      Sleep ( rand() % 2000 );
    if ( observer_ )
      send_signal ( this, SLEEPING );
      Sleep ( rand() % 2000 );
  }
  send_signal ( this, DEAD );
}
```

Für diese Codemodule ist eine Integration mit einer graphischen Benutzeroberfläche vorzunehmen.

B.4 Integration und Test

Nachdem die Softwarekomponenten ***Observer*** und ***Philosopher*** auf den Ausführungsmaschinen bereitgestellt wurden, können mit Hilfe des in $CORE_{WARE}$ enthaltenen generischen Konfigurationswerkzeuges verschiedene Konfigurationen des Softwaresystems aufgebaut und getestet werden. Dabei wird die durch $CORE_{MAP}$ unterstützte Konfigurierbarkeit der Softwarekomponenten ausgenutzt (vgl. Abb. 89).

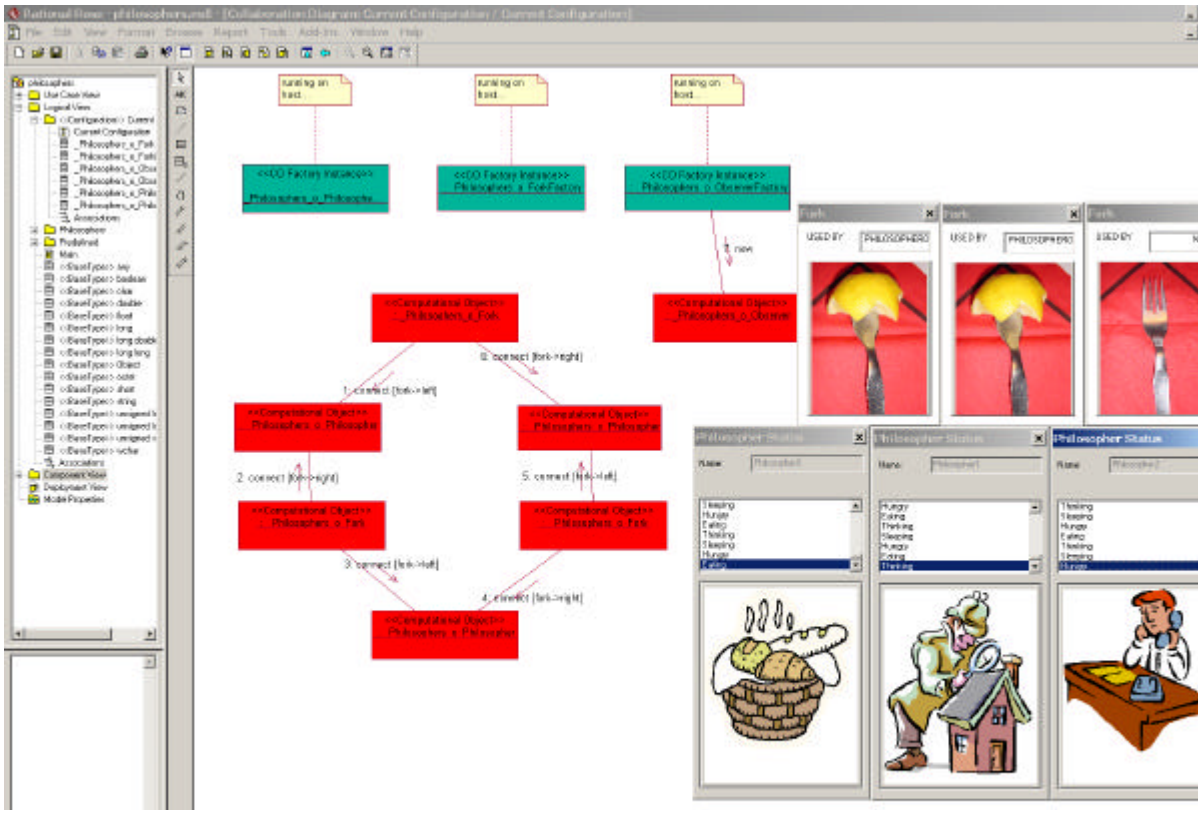


Abb. 89 Integration und Test

ANHANG C

Syntax von eODL

```
<specification> ::= <definition>+
<definition> ::= <type_dcl> ";"
            | <const_dcl> ";"
            | <except_dcl> ";"
            | <interface> ";"
            | <object_template> ";"
            | <artifact> ";"
            | <module> ";"
            | <value> ";"
            | <signal_dcl> ";"
            | <mediaset_dcl> ";"
            | <mediatype_dcl> ";"
            | <medium_dcl> ";"
            | <component_dcl> ";"
<module> ::= "module" <identifier> "{" <definition> + "}"
<component_dcl> ::= "component" <identifier> "realizes" "(" <realizes_dcl> ")"
<realizes_dcl> ::= <scoped_name> { "," <scoped_name> }*
<object_template> ::= <object_template_header> "{" <object_template_export> "}"
<object_template_header> ::= "CO" <identifier> [ <object_inheritance_spec> ]
<object_inheritance_spec> ::= ":" <scoped_name> { "," <scoped_name> }*
<object_template_export> ::= <object_export>*
<object_export> ::= <export>
            | <reqrd_interf_templates> ";"
            | <suptd_interf_templates> ";"
            | <use_dcl> ";"
            | <provide_dcl> ";"
            | <implements_dcl> ";"
            | <state_def_dcl> ";"
<reqrd_interf_templates> ::= "requires" <scoped_name> { "," <scoped_name> }*
```

```
<supd_interf_templates> ::= “supports“ <scoped_name> { “,“ <scoped_name> }*
<use_dcl> ::= “use“ [ “multiple“ ] <scoped_name> <identifier>
<provide_dcl> ::= “provide“ [ “multiple“ ] <scoped_name> <identifier>
<artifact> ::= <artifact_dcl>
            | <artifact_forward_dcl>
<artifact_forward_dcl> ::= “artifact“  <identifier>
<artifact_dcl> ::= <artifact_header> “{“ <artifact_body> “}“
<artifact_header> ::= “artifact“ <identifier> [ <artifact_inheritance_spec> ]
<artifact_inheritance_spec> ::= “:“ <scoped_name> { “,“ <scoped_name> }*
<artifact_body> ::= [ <impl_elem_dcl>* ]
<impl_elem_dcl> ::= <identifier> “implements“ <impl_case_dcl> <scoped_name> “;“
<impl_case_dcl> ::= “supply“ | “use“
<implements_dcl> ::= “implemented“ “by“ <artifact_with_policy>
            { “,“ <artifact_with_policy>* }
<artifact_with_policy> ::= <scoped_name> [ “with“ <instantiation_policy_dcl> ]
<instantiation_policy_dcl> ::= “ArtifactPool“
            | “ArtifactPerRequest“
            | “Singleton“
            | “UserDefined“
<state_def_dcl> ::= “state“ <scoped_name> [ “provided“ “to“ “(“ <provided_to_dcl> “)“ ]
<provided_to_dcl> ::= <scoped_name> { “,“ <scoped_name> }*
<interface> ::= <interface_dcl>
            | <forward_dcl>
<interface_dcl> ::= <interface_header> “{” <interface_body> “}”
<forward_dcl> ::= [ “abstract” ] “interface” <identifier>
<interface_header> ::= [ “abstract” ] “interface”
            <identifier> [ <interface_inheritance_spec> ]
<interface_body> ::= <export> *
<export> ::= <type_dcl> “;”
            | <const_dcl> “;”
            | <except_dcl> “;”
            | <attr_dcl> “;”
            | <op_dcl> “;”
            | <produce_dcl> “;“
            | <consume_dcl> “;“
            | <source_dcl> “;“
            | <sink_dcl> “;“
<produce_dcl> ::= “produce“ <scoped_name> <identifier>
<consume_dcl> ::= “consume“ <scoped_name> <identifier>
<source_dcl> ::= “source“ <scoped_name> <identifier>
<sink_dcl> ::= “sink“ <scoped_name> <identifier>
<interface_inheritance_spec>::=“:” <interface_name> { “,” <interface_name> } *
<interface_name> ::= <scoped_name>
<scoped_name> ::= <identifier>
            |“::” <identifier>
            | <scoped_name> “::” <identifier>
<signal_dcl> ::= “signal“ <identifier> “{“ <member_list> “}“
<mediaset_dcl> ::= “mediaset“ <identifier> “{“ <member_list> “}“
<mediatype_dcl> ::= “mediatype“ <identifier>
<medium_dcl> ::= “medium“ <identifier> “(“ <scoped_name> { “,“ <scoped_name> }* “)“
<value> ::= ( <value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)
<value_forward_dcl> ::= [ “abstract” ] “valuetype” <identifier>
<value_box_dcl> ::= “valuetype” <identifier> <type_spec>
<value_abs_dcl> ::= “abstract” “valuetype” <identifier> [ <value_inheritance_spec> ]
            “{“ <export>* “}”
```

<value_dcl> ::= <value_header> “{“ < value_element>* “}”
<value_header> ::= [“custom”] “valuetype” <identifier> [<value_inheritance_spec>]
<value_inheritance_spec> ::= [“:” [“truncatable”] <value_name>
{ “,” <value_name> }*]
[“supports” <interface_name>
{ “,” <interface_name> }*]
<value_name> ::= <scoped_name>
<value_element> ::= <export> | < state_member> | <init_dcl>
<state_member> ::= (“public” | “private”)
<type_spec> <declarators> “;”
<init_dcl> ::= “factory” <identifier> “(“ [<init_param_decls>] “)” “;”
<init_param_decls> ::= <init_param_decl> { “,” <init_param_decl> }
<init_param_decl> ::= <init_param_attribute> <param_type_spec> <simple_declarator>
<init_param_attribute> ::= “in”
<const_dcl> ::= <const_dcl>::=“const” <const_type>
<identifier> “=” <const_exp>
<const_type> ::= <integer_type>
| <char_type>
| <wide_char_type>
| <boolean_type>
| <floating_pt_type>
| <string_type>
| <wide_string_type>
| <fixed_pt_const_type>
| <scoped_name>
| <octet_type>
<const_exp> ::= <or_expr>
<or_expr> ::= <xor_expr> | <or_expr> “|” <xor_expr>
<xor_expr> ::= <and_expr> | <xor_expr> “^” <and_expr>
<and_expr> ::= <shift_expr> | <and_expr> “&” <shift_expr>
<shift_expr> ::= <add_expr>
| <shift_expr> “>>” <add_expr>
| <shift_expr> “<<”<add_expr>
<add_expr> ::= <mult_expr>
| <add_expr> “+” <mult_expr>
| <add_expr> “-” <mult_expr>
<mult_expr> ::= <unary_expr>
| <mult_expr> “*” <unary_expr>
| <mult_expr> “/” <unary_expr>
| <mult_expr> “%” <unary_expr>
<unary_expr> ::= <unary_operator> <primary_expr> | <primary_expr>
<unary_operator> ::= “-”|“+”|“~”
<primary_expr> ::= <scoped_name> | <literal> | “(” <const_exp> “)”
<literal> ::= <integer_literal>
| <string_literal>
| <wide_string_literal>
| <character_literal>
| <wide_character_literal>
| <fixed_pt_literal>
| <floating_pt_literal>
| <boolean_literal>
<boolean_literal> ::= “TRUE”|“FALSE”
<positive_int_const> ::= <const_exp>
<type_dcl> ::= “typedef” <type_declarator>
| <struct_type>| <union_type>

```
            | <enum_type>
            | “native” <simple_declarator>
<type_declarator> ::= <type_spec> <declarators>
<type_spec> ::= <simple_type_spec> | <constr_type_spec>
<simple_type_spec> ::= <base_type_spec>
            | <template_type_spec>
            | <scoped_name>
<base_type_spec> ::= <floating_pt_type>
            | <integer_type>
            | <char_type>
            | <wide_char_type>
            | <boolean_type>
            | <octet_type>
            | <any_type>
            | <object_type>
            | <value_base_type>
<template_type_spec> ::= <sequence_type>
            | <string_type>
            | <wide_string_type>
            | <fixed_pt_type>
<constr_type_spec> ::= <struct_type>
            | <union_type>
            | <enum_type>
<declarators> ::= <declarator> { “,” <declarator> } *
<declarator> ::= <simple_declarator> | <complex_declarator>
<simple_declarator> ::= <identifier>
<complex_declarator> ::= <array_declarator>
<floating_pt_type> ::= “float”
            | “double”
            | “long” “double”
<integer_type> ::= <signed_int> | <unsigned_int>
<signed_int> ::= <signed_short_int>
            | <signed_long_int>
            | <signed_longlong_int>
<signed_short_int> ::= “short”
<signed_long_int> ::= “long”
<signed_longlong_int> ::= “long” “long”
<unsigned_int> ::= <unsigned_short_int>
            | <unsigned_long_int>
            | <unsigned_longlong_int>
<unsigned_short_int> ::= “unsigned” “short”
<unsigned_long_int> ::= “unsigned” “long”
<unsigned_longlong_int> ::= “unsigned” “long” “long”
<char_type> ::= “char”
<wide_char_type> ::= “wchar”
<boolean_type> ::= “boolean”
<octet_type> ::= “octet”
<any_type> ::= “any”
<object_type> ::= “Object”
<struct_type> ::= “struct” <identifier> “{” <member_list> “}”
<member_list> ::= <member>+
<member> ::= <type_spec> <declarators> “;”
<union_type> ::= “union” <identifier> “switch”
            “(” <switch_type_spec> “)”
            “{” <switch_body> “}”
```

<switch_type_spec> ::= <integer_type>
| <char_type>
| <boolean_type>
| <enum_type>
| <scoped_name>
<switch_body> ::= <case>+
<case> ::= <case_label>+ <element_spec> “;”
<case_label> ::= “case” <const_exp> “:”|“default” “:”
<element_spec> ::= <type_spec> <declarator>
<enum_type> ::= “enum” <identifier> “{” <enumerator> { “,” <enumerator> } * “}”
<enumerator> ::= <identifier>
<sequence_type> ::= “sequence” “<” <simple_type_spec> “,”
<positive_int_const> “>” | “sequence” “<” <simple_type_spec> “>”
<string_type> ::= “string” “<” <positive_int_const> “>” | “string”
<wide_string_type> ::= “wstring” “<” <positive_int_const> “>” | “wstring”
<array_declarator> ::= <identifier> <fixed_array_size>+
<fixed_array_size> ::= “[” <positive_int_const> “]”
<attr_dcl> ::= [“readonly”] “attribute”
<param_type_spec> <simple_declarator> { “,” <simple_declarator> }*
<except_dcl> ::= “exception” <identifier> “{“ <member>* “}”
<op_dcl> ::= [<op_attribute>] <op_type_spec>
<identifier> <parameter_dcls> [<raises_expr>] [<context_expr>]
<op_attribute> ::= “oneway”
<op_type_spec> ::= <param_type_spec> |“void”
<parameter_dcls> ::= “(” <param_dcl> { “,” <param_dcl> } * “)”|“(” “)”
<param_dcl> ::= <param_attribute> <param_type_spec> <simple_declarator>
<param_attribute> ::= “in”|“out”|“inout”
<raises_expr> ::= “raises” “(” <scoped_name>{ “,” <scoped_name> } * “)”
<context_expr> ::= “context” “(” <string_literal>{ “,” <string_literal> } * “)”
<param_type_spec> ::= <base_type_spec>
| <string_type>
| <wide_string_type>
| <scoped_name>
<fixed_pt_type> ::= “fixed” “<“ <positive_int_const> “,”<positive_int_const> “>”
<fixed_pt_const_type> ::= “fixed”
<value_base_type> ::= “ValueBase”

CoRE - Komponentenorientierte Entwicklung offener verteilter Softwaresysteme im Telekommunikationskontext

Band III - Plattformunterstützung und Ableitungsregeln für Softwarekomponenten

Olaf Kath

Humboldt-Universität zu Berlin
Institut für Informatik
Rudower Chaussee 25
12489 Berlin

kath@informatik.hu-berlin.de
Tel. +49 (30) 2093 3117
Fax. +49 (30) 2093 3112

VORWORT

Grundanliegen von *CoRE* ist die Integration zweier Entwicklungstechniken im Bereich der Telekommunikation in Gestalt der Objektorientierte Modellierung einerseits und des Einsatzes von Komponentenarchitekturen andererseits mit dem vordergründigen Ziel, eine automatische Ableitung von Softwarekomponenten eines zu entwickelnden verteilten Softwaresystems aus Entwurfsmodellen zu unterstützen. Die Präzisierung der automatischen Ableitung von Softwarekomponenten erfolgt in *CoRE* durch die Definition geeigneter Ableitungsregeln, die objektorientierte Modellierungskonzepte von $CoRE_{CEPT}$ auf Mechanismen einer Komponentenarchitektur abbilden. Auf der Basis dieser Ableitungsregeln lassen sich Entwicklungswerkzeuge realisieren, die objektorientierte Modelle als Resultat der Entwurfsphase in Softwarekomponenten überführen.

Der Einsatz von Komponentenarchitekturen zur Bereitstellung der Infrastruktur für die Interaktion der Softwarekomponenten von Softwaresystemen hat sich in den letzten Jahren zunehmend durchgesetzt. Eine Analyse der *Gartner Group* ergab, daß 80% der Unternehmen in der Softwareindustrie bereits Komponentenarchitekturen wie CORBA [OMG CORBA] oder COM [MS COM] einsetzen oder deren Einsatz mittelfristig planen. Komponentenarchitekturen realisieren eine Infrastruktur, die die Integration von Softwarekomponenten verteilter Softwaresysteme stark vereinfacht. Die Telekommunikation als ein softwareintensiver Bereich setzt ebenfalls Komponentenarchitekturen ein, stellt aber an diese spezifische Anforderungen wie Skalierbarkeit, Konfigurierbarkeit, Unterstützung von *Continuous-Media*-Interaktionen und Ausfallsicherheit. Der Einsatz von Komponentenarchitekturen ist eine wesentliche Entwicklungstechnik im Kontext von Entwicklungsprozessen für verteilte Telekommunikationssoftwaresysteme.

Es ist jedoch festzustellen, daß eine Integration dieser Entwicklungstechnik mit weiteren, ebenso verbreiteten Entwicklungstechniken wie objektorientierte Modellierung bisher unbefriedigend gelöst ist. Insbesondere ist die Unterstützung des Übergangs von der Entwurfsphase in die nachfolgenden Entwicklungsphasen durch Entwicklungswerkzeuge nicht zufriedenstellend. Die Verfügbarkeit solcher Werkzeuge ist aber vielfach ein Entscheidungskriterium für oder gegen die Verwendung konkreter Entwicklungstechniken.

In diesem Band wird zunächst $CoRE_{WARE}$, eine auf CORBA basierende erweiterte Komponentenarchitektur, konzipiert. Die Notwendigkeit dafür ergibt sich aus der Tatsache, daß derzeit eingesetzte Komponenten-

architekturen zwar eine Infrastruktur für die Interaktion zwischen Softwarekomponenten realisieren, jedoch nicht darüber hinausgehende Anforderungen von Telekommunikationssoftwaresystemen *a priori* erfüllen. Diese spezifischen Anforderungen werden in dem vorliegenden Band ausgehend von der Diskussion der in [CoRE I] formulierten allgemeinen Anforderungen an Entwicklungstechniken für solche Softwaresysteme hergeleitet, sie präzisieren die allgemeinen Anforderungen in [CoRE I], Abschnitt 1.5 bezüglich der zum Einsatz kommenden Technologien.

Die in $CoRE_{WARE}$ realisierten Mechanismen werden als Ziel der Ableitungsregeln für Softwarekomponenten genutzt. Diese Ableitungsregeln, zusammengefaßt in $CoRE_{MAP}$, werden auf $CoRE_{CEPT}$ basierend entwickelt und dargestellt. Dazu wird das in [CoRE II] beschriebene Metamodell herangezogen: Es werden Ableitungsregeln für *alle* im Metamodell von $CoRE_{CEPT}$ enthaltenen Konzepte und deren Beziehungen erstellt. Für jede dieser Ableitungsregeln werden mittels der Notationen von $CoRE_{TATIONS}$ Beispiele zur Illustration dieser Regeln gegeben. Die Wirkungsweise der durch die Ableitungsregeln generierten Codemodule wird durch Interaktionsdiagramme (UML-Sequenznotation) illustriert. Bei der Definition dieser Ableitungsregeln werden allgemein anerkannte Implementierungsmuster (z.B. [GHJ+ 99]) verwendet, um die Nachvollziehbarkeit des Regelwerkes zu erhöhen sowie einem Entwickler das Verständnis der produzierten Codefragmente zu erleichtern.

Durch den Autor ist eine vollständige Implementierung von $CoRE_{WARE}$ und $CoRE_{MAP}$ unter Benutzung der Realisierungen von $CoRE_{CEPT}$ und $CoRE_{TATIONS}$ erstellt worden. Diese Implementierung dient der Überprüfung der Ableitungsregeln selbst, aber auch der Praktikabilität des Konzeptraumes: Alle Elemente des Konzeptraumes wurden durch die Anwendung dieses Prinzips verifiziert, notwendige Erweiterungen des Konzeptraumes aus dieser Verifikation gefolgert. Die Implementierung liegt in Form von einsetzbaren Entwicklungswerkzeugen vor, die mit dem Produkt Rational Rose integriert wurden. Entwickler, die diese Werkzeuge einsetzen, können vorliegende objektorientierte Entwurfsmodelle automatisch in Softwarekomponenten übertragen. Die dann verbleibende Entwicklungsaufgabe ist ausschließlich die Komplettierung der Codemodule der Softwarekomponenten durch Einfügen von *Business Logic*. In einer Fallstudie wird in diesem Band der dazu notwendige Aufwand illustriert.

Es wurden folgende Schriftarten eingesetzt:

- ***Elemente von Modellen sind in "Arial Italic" notiert,***
- **Beispiele für produzierte CORBA-IDL-Definitionen werden in "Arial Bold" verfaßt,**
- **`Beispiele für produzierte C++-Definitionen sind in "Courier New Bold" und`**
- *englische Begriffe, für die keine adäquate Übersetzung gefunden wurde, sind kursiv geschrieben.*

Berlin, im Juli 2001

Olaf Kath
Humboldt-Universität zu Berlin
Institut für Informatik
Rudower Chaussee 25
12489 Berlin
kath@informatik.hu-berlin.de

INHALT

KAPITEL 1

Einführung in CoRE$_{MAP}$ und CoRE$_{WARE}$

Ein wesentliches Ziel dieser Arbeit besteht neben der Definition des Konzeptraumes und einer die identifizierten Konzepte widerspiegelnden Notation in der Definition von Ableitungsregeln (*CoRE$_{MAP}$*) für Softwarekomponenten. Diese Ableitungsregeln definieren die Abbildung der Konzepte von *CoRE$_{CEPT}$* auf Basiskonstrukte einer Komponentenarchitektur (*CoRE$_{WARE}$*). Die Definition dieser Ableitungsregeln wird die Realisierung von Entwurfsmodellen, die mittels *CoRE* entworfen wurden, in konkreten verteilten Umgebungen erlauben.

In der Literatur [EU P910][EU P715] werden bezüglich des Zusammenhangs zwischen Komponentenarchitektur und *Deployment*- und Ausführungsunterstützung für Softwarekomponenten oftmals die Termini Serviceplattform, *Component-Support*-Plattform und *Distributed-Processing*-Umgebung erwähnt, deren Zusammenhänge jedoch abweichend präsentiert.

In *CoRE$_{WARE}$* bildet die *Distributed-Processing*-Umgebung die technologische Grundlage zur Unterstützung der Interaktion von Objekten eines verteilten Softwaresystems (s. Abb. 1). Allgemeine Objektdienste - insbesondere zum Auffinden von Objekten - werden ebenfalls durch die *Distributed-Processing*-Umgebung bereitgestellt. Diese Umgebung ist unter Verwendung von CORBA-2.4-Produkten in *CoRE$_{WARE}$* realisiert.

Component-Support-Plattformen nutzen die durch die *Distributed-Processing*-Umgebung bereitgestellte Funktionalität. Sie stellen *Deployment*- und Ausführungsunterstützung für Softwarekomponenten bereit. Während eine *Distributed-Processing*-Umgebung, die in *CoRE$_{WARE}$* unter Nutzung von CORBA-Produkten realisiert ist, nur Objektinteraktion bzgl. eines angebotenen bzw. genutzten Interfaces eines COs ermöglicht, stellt eine *Component-Support*-Plattform zur Ausführungszeit den gemeinsamen Kontext aller Interfaces eines COs her, die durch dieses angeboten bzw. genutzt werden. Sie realisiert Zustand, Identität und Verhalten von COs. Des weiteren bietet sie Mechanismen zur Integration, Konfiguration und Inbetriebnahme der Softwarekomponenten als Zusammenfassung der das Verhalten eines oder mehrerer COs realisierenden Artefakte. Kommerzielle Beispiele für *Component-Support*-Plattformen sind EJB (*Enterprise Java Beans*) und CORBA *Components*, beide können auf der Basis einer mit CORBA-2.4-Produkten realisierten *Distributed-Processing*-Umgebung bereitgestellt werden.

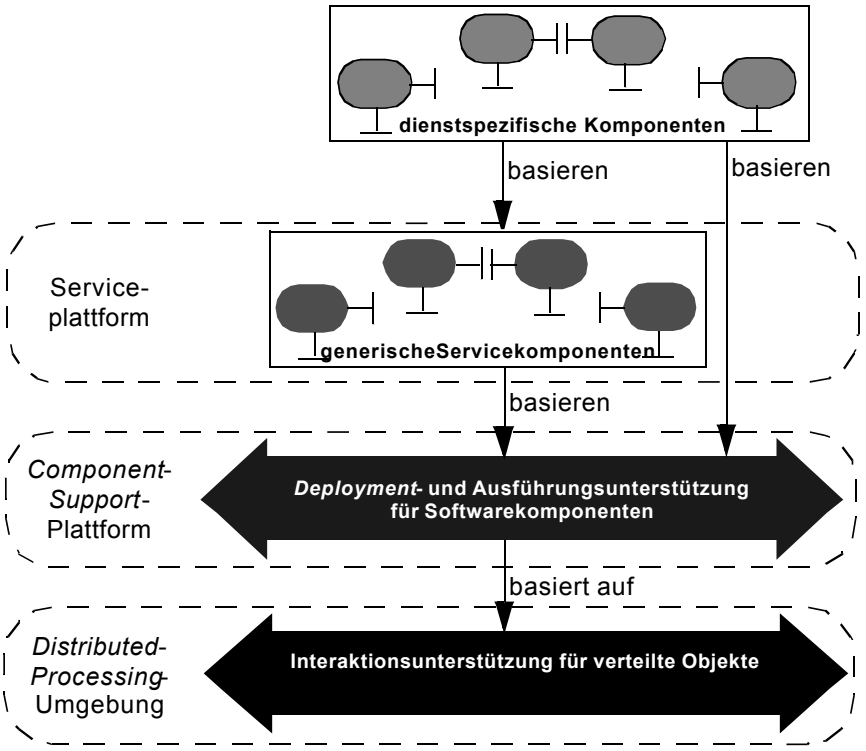


Abb. 1 Distributed-Processing-Umgebung, Komponenten- und Diensteplattform

Eine Serviceplattform ist eine Komposition vordefinierter, in einer bestimmten Domäne allgemein nutzbarer Softwarekomponenten, die die Dienste der *Component-Support*-Plattform nutzen. Während *Component-Support*-Plattformen weitgehend unabhängig von der konkreten Anwendungsdomäne sind, werden Serviceplattformen, d.h. die Menge der in einer konkreten Domäne allgemein nutzbaren Softwarekomponenten domänenspezifisch entworfen. Eine solche Menge von Softwarekomponenten für die Domäne Telekommunikation spezifiziert beispielsweise die TINA-Servicearchitektur in der Telekommunikationsdomäne [TINASA]. Auf der Basis dieser allgemeinen Softwarekomponenten werden konkrete, anwendungsspezifische Softwarekomponenten definiert.

Softwarekomponenten, die auf der *Component-Support*-Plattform basieren, entstehen durch Entwicklungswerkzeuge, die Ableitungsregeln umsetzen. Ziel der Anwendung von Ableitungsregeln ist die Erzeugung aller Fragmente, deren Zusammensetzung eine ausführbare Softwarekomponente mit deren Interfaces ergibt. In Abhängigkeit von der benutzten *Component-Support*-Plattform und der dieser zugrunde liegenden *Distributed-Processing*-Umgebung sind diese Fragmente:

- *Component-Support*-Plattform-spezifische Interfacebeschreibungen der Softwarekomponenten (abgeleitet aus definierten Interfacetypen und CO-Typen in einem auf *CoRE* basierenden objektorientierten Entwurfsmodell),
- programmiersprachliche Konstrukte für die Artefakte eines Entwurfsmodells,
- programmiersprachliche Konstrukte für die Einbindung der Mechanismen der *Component-Support*-Plattform in die entstehenden Softwarekomponenten,
- *Component-Support*-Plattform-spezifische Repräsentation von Bindungsmechanismen (*Policies*, Prädikate und Bindungsregeln),
- *Make*-Files zur Kompilation und zum Binden maschinenspezifischer Codemodule (falls erforderlich),
- Deskriptoren für automatisierte *Deployment*-Abläufe.

CORE$_{MAP}$ stellt dabei sicher, daß in einer Softwarekomponente ausschließlich diejenigen Codemodule enthalten sind, die zur Realisierung derjenigen CO-Typen erforderlich sind, für die eine *Realize*-Relation zu dieser Softwarekomponente angegeben wurde.

1.1 Anforderungengsanalyse

In [CoRE I], Abschnitt 1.5 wurden allgemeine Anforderungen formuliert, die an Techniken zur Entwicklung offener, verteilter Telekommunikationssoftwaresysteme gestellt werden. Diese Anforderungen können nun für die Entwicklungstechnik *CORE*$_{MAP}$ zur automatischen Ableitung von Softwarekomponenten und die dabei Verwendung findende Komponentenarchitektur *CORE*$_{WARE}$ präzisiert werden. Dabei sind zum einen Verfeinerungen der allgemeinen Anforderungen bezüglich der *Component-Support*-Plattform von *CORE*$_{WARE}$ vorzunehmen, die Zielumgebung der Generierung von Softwarekomponenten ist. Zum anderen werden Verfeinerungen der allgemeinen Anforderungen im Rahmen der Codegenerierung von *CORE*$_{MAP}$ selbst untersucht.

Grundsätzlich stellt eine *Component-Support*-Plattform die Basisunterstützung für die Ausführung von Softwarekomponenten dar, in denen Codemodule zusammengefaßt sind, die CO-Typen realisieren. Dies wird i.allg. durch die Bereitstellung einer Kommunikations- und Ausführungsinfrastruktur für Softwarekomponenten erreicht. Die Anbindung von Softwarekomponenten an diese Infrastruktur erfolgt via Interfaces, die durch die in Softwarekomponenten enthaltenen Codemodule spezialisiert und realisiert bzw. genutzt werden.

1.1.1 Offenheit

Distributed-Processing-Umgebungen stellen die Basisinfrastruktur zur Interaktion der Softwarekomponenten verteilter Softwaresysteme bereit. Die *Component-Support*-Plattform in *CORE*$_{WARE}$ erweitert diese Infrastruktur um Mechanismen, die eine einheitliche Sicht auf Softwarekomponenten ermöglichen. Diese einheitliche Sicht umfaßt insbesondere das Management der in diesen Softwarekomponenten enthaltenen Codemodule, die CO-Typen in Entwurfsmodellen repräsentieren.

(1) KONFIGURATIONSMANAGEMENT UND NAVIGATION. Ein CO-Typ in *CORE* repräsentiert einen Kontext der Bereitstellung bzw. der Nutzung von Interfaces. Das *Port*-Konzept wurde in *CORE*$_{CEPT}$ zur Definition dieses Kontextes bezüglich eines einzelnen Interfacetyps eingeführt, CO-Typen können eine oder mehrere *Port*-Definitionen besitzen. Die Mechanismen von *CORE*$_{WARE}$ und die auf diesen Mechanismen basierende Ableitung von Softwarekomponenten durch *CORE*$_{MAP}$ müssen *Port*-Definitionen in Entwurfsmodellen auf allgemeine Managementinterfaces der Repräsentationen von CO-Typen in der *Component-Support*-Plattform abbilden. Diese Abbildung muß in einer Art und Weise vorgenommen werden, daß Werkzeuge entwickelt werden können, mit deren Hilfe verteilte Softwaresysteme homogen konfigurierbar und steuerbar sind. Zu diesem Zweck muß die Repräsentation eines CO-Typs, die durch die Ableitung von Softwarekomponenten aus Entwurfsmodellen entsteht, Interfaces bereitstellen, die die Navigation (innerhalb des Kontextes eines CO-Typs) von einem CO zu seinen genutzten bzw. bereitgestellten Interfaces erlauben. Darüber hinaus sind generische und CO-Typ-spezifische Interfaces zu realisieren, die Repräsentationen konkreter COs mit den Kommunikationskanälen der *Component-Support*-Plattform koppeln.

(2) REFLEXION VON ENTWURFSINFORMATION. *CORE*$_{MAP}$ soll sicherstellen, daß zur Ausführungszeit auf alle Entwurfsinformationen von COs zugegriffen werden kann, deren Repräsentation in Softwarekomponenten enthalten ist. Diese Anforderung schließt nicht nur Informationen über die extern sichtbare Struktur (CO-Typ, *Port*-Definitionen und genutzte bzw. bereitgestellte Interfaces) ein, sondern auch die Interna der Realisierung des CO-Typs durch in Entwurfsmodellen definierte Artefakte. Ist diese Anforderung erfüllt, können

Werkzeuge auch ohne Kenntnis des Entwurfsmodells verteilte Softwaresysteme, die unter Anwendung von *CoRE* entstanden sind, konfigurieren und steuern.

(3) UNABHÄNGIGKEIT DER ARTEFAKTREPRÄSENTATION VON *CoRE$_{WARE}$*. Artefaktrepräsentationen als programmiersprachliche Konstrukte werden durch Ableitungsregeln aus Artefaktdefinitionen entsprechender Entwurfsmodelle erzeugt. Artefaktrepräsentationen werden durch den Entwickler um programmiersprachliche Konstrukte erweitert, die die Realisierung des Verhaltens von COs erbringen. Diese programmiersprachlichen Konstrukte bilden damit - neben Entwurfsmodellen - entscheidende Bausteine der Wiederverwendung. Dementsprechend wird gefordert, daß die Repräsentation von Artefakten durch programmiersprachliche Konstrukte unabhängig von konkreten Mechanismen einer *Component-Support*-Plattform ist. Es müssen also in *CoRE$_{MAP}$* Ableitungsregeln bereitgestellt werden, die die plattformunabhängige Repräsentation von Artefakten an die Mechanismen einer *Component-Support*-Plattform koppeln.

1.1.2 Konzeptionelle Offenheit

Eine wesentliche allgemeine Anforderung ist die konzeptionelle Offenheit. Diese Forderung adressiert die Offenheit der Entwicklungstechniken selbst für Erweiterungen bzw. Anpassungen an konkrete Einsatzumgebungen.

(4) ERWEITERBARKEIT VON *CoRE$_{MAP}$*. Erweiterung bzw. Anpassung der Entwicklungstechniken umfaßt zum einen die Möglichkeit der Modifikation des Konzeptraumes von *CoRE* (und damit der konkreten Notationen für Entwurfsmodelle). Zum anderen müssen Erweiterungen bzw. Anpassungen von *CoRE$_{CEPT}$* Veränderungen der Ableitungsregeln für Softwarekomponenten im Rahmen von *CoRE$_{MAP}$* bewirken, die auf einfache Art und Weise dem Regelwerk zur automatischen Ableitung hinzugefügt werden können bzw. bestehende Regeln in diesem Regelwerk verändern.

1.1.3 Unterstützung aller relevanten Interaktionsarten

CoRE gestattet die Modellierung von Interaktionselementen unterschiedlicher Interaktionsarten im Kontext eines Interfacetyps. Die *Component-Support*-Plattform *CoRE$_{WARE}$* und das auf dieser basierende Regelwerk *CoRE$_{MAP}$* zur Ableitung von Softwarekomponenten müssen diesen gemeinsamen Kontext repräsentieren.

(5) BEREITSTELLUNG VON MECHANISMEN ZUR UNTERSTÜTZUNG DER INTERAKTIONSARTEN. Diese Anforderung impliziert, daß die *Component-Support*-Plattform von *CoRE$_{WARE}$* Mechanismen bereitstellen muß, die die Interaktionsarten Signalinteraktion, *Continuous-Media*-Interaktion und operationale Interaktion unterstützen. Diese Mechanismen müssen durch die Regeln zur Ableitung von Softwarekomponenten genutzt werden. Sie werden so definiert, daß sie die durch die *Distributed-Processing*-Umgebung bereitgestellte Infrastruktur ausnutzen.

1.1.4 Flexible Skalierbarkeit

Die allgemeine Anforderung an die Flexibilität der Skalierbarkeit der durch *CoRE* produzierten Softwarekomponenten impliziert Anforderungen an *CoRE$_{MAP}$* bezüglich der Ableitungsregeln. Die Grundlage für die Erfassung von Flexibilitätsanforderungen in Entwurfsmodellen ist durch die Definition der Instanziierungsmuster für Artefakte in *CoRE$_{CEPT}$* gegeben. Die Aufgabe von *CoRE$_{MAP}$* besteht in der Repräsentation dieser Instanziierungsmuster in programmiersprachlichen Konstrukten.

(6) REPRÄSENTATION DER INSTANZIIERUNGSMUSTER IN ABGELEITETEN SOFTWAREKOMPONENTEN. Im Rahmen der automatischen Ableitung von Softwarekomponenten aus Entwurfsmodellen sollten die durch *CoRE$_{CEPT}$* definierten Instanziierungsmuster für Artefakte unter Anwendung allgemein anerkannter Entwurfsmuster [GHJ+ 99] bei der Erzeugung von programmiersprachlichen Konstrukten durch *CoRE$_{MAP}$* abgebildet wer-

den. Des weiteren müssen die Ableitungsregeln so formuliert sein, daß die Erfüllung von Skalierbarkeitsanforderungen an resultierende Softwarekomponenten mittels Experimenten bereits überprüfbar sind, bevor überhaupt eine konkrete, programmiersprachliche Realisierung der Artefakte vorliegt.

(7) KOPPLUNG DER ARTEFAKTREALISIERUNGEN AN DIE *Component-Support*-PLATTFORM. Die allgemeine Anforderung der Offenheit an die Ableitungsregeln für Softwarekomponenten impliziert die Anforderung der Ableitung von programmiersprachlichen Konstrukten, der die plattformunabhängige Repräsentation von Artefakten an spezifische Mechanismen einer *Component-Support*-Plattform koppelt. Diese Anbindung wird i.allg. durch Anwendung des Entwurfsmusters *Delegation* [GHJ+ 99] erreicht. Bei der Anwendung dieses Entwurfsmusters sind die Ableitungsregeln so zu formulieren, daß diese Anbindung möglichst performant gestaltet wird: Die Anzahl der notwendigen Kopien der Parameter eines Operationsrufes bei der Delegierung ist zu minimieren, die Ziele der Delegierung sind unbedingt nicht-polymorph zu definieren.

1.1.5 Flexible Adaptierbarkeit

Artefakte realisieren das Verhalten von CO-Typen, in Entwurfsmodellen werden *Implements*-Relationen zwischen konkreten CO-Typen und Artefakten definiert. Diese Relationen sollen durch Ableitungsregeln von $CoRE_{MAP}$ so in Softwarekomponenten repräsentiert werden, daß das Prinzip der Komponierbarkeitsflexibilität (vgl. [CoRE I], Abschnitt 3.1.18) unterstützt wird.

(8) FLEXIBLE UMSETZUNG DER RELATIONEN ZWISCHEN CO-TYPEN UND ARTEFAKTEN. Es wurde bereits festgestellt, daß die konkreten Implementierungen von Atefaktrepräsentationen durch Entwickler neben der Spezifikation des gesamten Softwaresystems wichtige Bausteine der Wiederverwendung darstellen. Es ist zu fordern, daß die Relationen zwischen Artefakten und CO-Typen so abgebildet werden, daß die Abhängigkeiten zwischen den Repräsentationen dieser Artefakte und den Repräsentationen von CO-Typen minimal sind. Diese Minimalitätsforderung schließt nicht nur die Repräsentation der CO-Typen selbst ein, sondern auch die Repräsentationen der Typen der von diesen bereitgestellten bzw. genutzten Interfaces.

(9) MINIMALITÄT DER ABHÄNGIGKEITEN ZWISCHEN $CoRE_{WARE}$ UND $CoRE_{MAP}$. Eine wesentliche Zielstellung von *CoRE* besteht in der möglichst einfach durchführbaren Adaption der Ableitungsregeln zur Ableitung von Softwarekomponenten für verschiedene *Component-Support*-Plattformen. Insofern ist zu fordern, daß die Abhängigkeiten zwischen $CoRE_{MAP}$ und $CoRE_{WARE}$ gering sind.

1.1.6 Gütebeschreibung und Garantie

Die Erfassung von Dienstgüteeigenschaften und die Umsetzung dieser im Rahmen der abgeleiteten Softwarekomponenten spielen in der Domäne Entwicklung von verteilten Telekommunikationssoftwaresystemen eine zentrale Rolle.

(10) ABLEITUNGSREGELN FÜR DIE BESCHREIBUNG VON DIENSTGÜTEEIGENSCHAFTEN. Es wird gefordert, daß Dienstgüteeigenschaften, die in Entwurfsmodellen auf der Grundlage von $CoRE_{CEPT}$ erfaßt sind, durch Ableitungsregeln in Codemodulen von Softwarekomponenten repräsentiert sind. $CoRE_{MAP}$ soll Ableitungsregeln umfassen, die programmiersprachliche Konstrukte sowie Interfacedefinitionen erzeugen, mit deren Hilfe die Bestimmung von Bindungsfällen, die Moderation der Verhandlung von erwarteten Güteeigenschaften sowie deren Sicherung und Überwachung möglich ist.

(11) ADAPTIERBARKEIT DER ABLEITUNGSREGELN. Die Ableitungsregeln zur Verhandlung, Sicherung und Überwachung für Dienstgüteeigenschaften sollen so gestaltet sein, daß eine Adaption dieser Regeln für unterschiedliche *Component-Support*-Plattformen und deren Mechanismen bezüglich der Sicht auf Güteeigenschaften möglich ist.

1.1.7 Flexible Integration von Softwarekomponenten

Softwarekomponenten, die mit *CoRE$_{MAP}$* erzeugt wurden, sollen sich in verschiedenen bestehenden Infrastrukturen einsetzen lassen. Darüber hinaus sollen bereits vorhandene Softwarekomponenten, auf eine beliebige Art und Weise erzeugt wurden, mit den durch *CoRE$_{MAP}$* aus Entwurfsmodellen abgeleiteten Softwarekomponenten integriert werden. Um dies zu ermöglichen, müssen die im folgenden diskutierten Anforderungen erfüllt sein.

(12) UNABHÄNGIGKEIT DER ARTEFAKTREPRÄSENTATIONEN VON DEN *CoRE$_{WARE}$*-MECHANISMEN. Existierende Codemodule liegen gewöhnlich als programmiersprachliche Konstrukte in Programmbibliotheken vor. Diese Codemodule sollen genutzt werden können, um das Verhalten von CO-Typen zu realisieren, d.h. sie sollen als Implementierung der Implementierungselemente von Artefaktrepräsentationen nutzbar sein. Diese Art der Integration ist dann möglich, wenn Ableitungsregeln in *CoRE$_{MAP}$* für programmiersprachliche Konstrukte existieren, in die bestehende Implementierungen eingepaßt werden können. Darüber hinaus sollten diese programmiersprachlichen Konstrukte möglichst wenige Abhängigkeiten bezüglich der Mechanismen von *CoRE$_{WARE}$* aufweisen. Die Ableitungsregeln für Artefakte müssen dann so gestaltet sein, daß eine Integration mit bestehenden Codemodulen möglichst einfach realisierbar ist.

(13) MÖGLICHKEIT DER INTEGRATION PERSISTENTER SPEICHERTECHNOLOGIEN. In vielen Situationen ist es notwendig, die Repräsentation von Zustandsattributen eines CO-Typs in persistenten Speichern (z.B. Datenbanken, Dateisysteme) zu hinterlegen und den Artefaktrepräsentationen den Zugriff auf diese zu gestatten. Die Ableitungsregeln für Softwarekomponenten müssen dementsprechend eine einfache Integrationsmöglichkeit für persistente Speichertechnologien enthalten sowie den Zugang von Artefaktrepräsentationen zu den Zustandsattributen von CO-Typen ermöglichen.

(14) INTEGRATION VON VORHANDENEN SOFTWAREKOMPONENTEN. Softwarekomponenten, die nicht als programmiersprachliche Codemodule, sondern in ihrer binären Form vorliegen, bilden oft eine wichtige Basis der Realisierung von Softwaresystemen. Die Ableitungsregeln von *CoRE$_{MAP}$* müssen so definiert sein, daß diese Softwarekomponenten im Rahmen von durch *CoRE$_{MAP}$* entstehenden Softwarekomponenten wiederverwendet werden können.

1.1.8 Kurze Entwicklungszeiten

CoRE soll kurze Entwicklungszeiten für Softwarekomponenten eines zu entwickelnden verteilten Softwaresystems ermöglichen, dabei aber eine hohe Qualität dieser Softwarekomponenten sichern. Darüber hinaus muß die einfache Integration dieser Softwarekomponenten durch *CoRE$_{WARE}$* gewährleistet werden.

(15) WERKZEUGE ALS IMPLEMENTIERUNGEN VON *CoRE$_{MAP}$*. Um diese generellen Zielstellungen zu realisieren, ist die Bereitstellung von Entwicklungswerkzeugen unerläßlich, die die definierten Ableitungsregeln für Softwarekomponenten umsetzen. Insofern ist die Forderung nach der Implementierbarkeit der Menge dieser Ableitungsregeln aufzustellen.

(16) MODELLAKTUALISIERUNG UND *Round-Trip* FÜR ENTWICKLUNGSITERATIONEN. Werkzeuge, die *CoRE$_{MAP}$* implementieren, sollen neben der Realisierung der Ableitung von Softwarekomponenten aus Entwurfsmodellen auch das Importieren von programmiersprachlichen Konstrukten und Interfacedefinitionen, die außerhalb eines Entwurfsmodells vorgenommen wurden, ermöglichen. Mit den importierten Definitionen soll es gelingen, die Konsistenz von Entwurfsmodellen und durch den Entwickler hinzugefügten Konstrukten zu sichern.

1.1.9 Vollständigkeit

Eine grundlegende Zielsetzung dieser Arbeit besteht im Nachweis der praktischen Anwendbarkeit der in *CoRE* zusammengefaßten Entwicklungstechniken in ihrer Gesamtheit. Eine Voraussetzung für diesen Nachweis ist die Vollständigkeit der Ableitungsregeln bezüglich aller in [CoRE II] definierten Konzepte von $CoRE_{CEPT}$.

(17) VOLLSTÄNDIGKEIT DER REGELN ZUR ABLEITUNG VON SOFTWAREKOMPONENTEN. Es wird ein Regelwerk zur automatischen Ableitung von Softwarekomponenten gefordert, das Ableitungsregeln für *alle* durch $CoRE_{CEPT}$ definierten Konzepte mit deren Semantik enthält. Dabei ist insbesondere sicherzustellen, daß für die grundsätzlichen Konzepte des Interfacetyps als Interaktionskontext für verschiedene Interaktionsarten und des CO-Typs als Kontext für das Angebot bzw. die Nutzung dieser Interaktionskontexte eine Repräsentation im Rahmen der *Component-Support*-Plattform in $CoRE_{WARE}$ erreicht wird, die diese Kontexte widerspiegelt. Dies ist durch entsprechende Ableitungsregeln zu gewährleisten. Die im Rahmen der Repräsentation von CO-Typen produzierten programmiersprachlichen Konstrukte sollen darüber hinaus für einen Entwickler nachvollziehbar sein, konkret soll sich die Struktur der Definition eines CO-Typs in den unterschiedlichen Sichten von *CoRE* in der produzierten Repräsentation äquivalent widerspiegeln.

1.2 Existierende Ansätze

Im folgenden werden existierende Ansätze vorgestellt und in die eingeführte Terminologie *Distributed-Processing*-Umgebung, *Component-Support*-Plattform und Serviceplattform eingeordnet. Im Vordergrund stehen dabei Ansätze, die sich als Grundlage einer *Distributed-Processing*-Umgebung oder *Component-Support*-Plattform eignen. Dabei werden sowohl standardisierte Lösungen als auch proprietäre Technologien betrachtet. Lösungen, die auf *De-Facto*- oder *De-Jure*-Standards beruhen, versprechen natürlich eine größere Vielfalt an einsetzbaren Produkten, die Teile dieser Standards realisieren und - begründet in der Standardisierung der Schnittstellen dieser Produkte - miteinander integriert werden können. Andererseits existieren proprietäre Lösungen für nicht-standardisierte Konzepte. Es bleibt abzuwarten, inwieweit diese Lösungen einer erfolgreichen Standardisierung zugeführt werden können.

1.2.1 Common Object Request Broker Architecture

1.2.1.1 Charakterisierung

Im Rahmen von *Object Management Group* wurde die *Common-Object-Request-Broker*-Architektur (CORBA, [OMG CORBA]) entwickelt und standardisiert. Diese Architektur enthält die folgenden Bestandteile:

- *Object Request Broker* (ORB) stellt die Basisinfrastruktur für CORBA-Objekte bereit, die mit anderen CORBA-Objekten in einer verteilten Umgebung interagieren. Durch ORBs werden diese Interaktionen unabhängig vom Ort der interagierenden CORBA-Objekte vermittelt (*Location Transparency*). ORBs stellen damit die Grundlage für die Entwicklung von Anwendungen basierend auf verteilten CORBA-Objekten dar. Darüber hinaus sichern ORBs die Zusammenarbeit der Softwarekomponenten von Softwaresystemen in heterogenen Umgebungen (d.h. unter nicht ausschließlicher Verwendung von CORBA realisierten) und homogenen Umgebungen (d.h. unter ausschließlicher Verwendung von CORBA realisierten).
- Durch *Object*-Services werden Schnittstellen von allgemein nutzbaren Softwarekomponenten der Infrastruktur spezifiziert. *Object*-Services unterstützen Basisfunktionalität zur Nutzung und Realisierung verteilter Anwendungen, und zwar unabhängig von der konkreten Anwendungsdomäne. *Object*-Services sind grundsätzlich Bestandteil einer allgemeinen CORBA-Infrastruktur, als Beispiele seien hier *Naming*-Ser-

vice [OMG CORBANaS], *Event*-Service [OMG CORBA ES] und *Notification*-Service [OMG CORBANoS] genannt.

- *Common Facilties* enthält - ebenso wie *Object*-Services - Definitionen der Schnittstellen von Softwarekomponenten, die von anderen Softwarekomponenten eines verteilten Softwaresystems genutzt werden können. Im Gegensatz zu *Object*-Services sind sie nicht Bestandteil einer allgemeinen CORBA-Infrastruktur, da sie nicht deren Allgemeinheitsgrad besitzen - sie sind nicht für alle CORBA-Anwendungen fundamental.
- *Application Objects* sind CORBA-Objekte in verteilten Softwaresystemen, die die CORBA-Infrastruktur nutzen, um miteinander zu interagieren. Diese Bestandteile werden i.allg. nicht der Standardisierung innerhalb der *Object Management Group* zugeführt.

CORBA definiert ein Objektmodell, das die Semantik eines CORBA-Objektes, dessen Interface und dessen Realisierung beschreibt. Zur Definition der Interfaces wird die Beschreibungssprache *Interface Definition Language* [OMG CORBA IDL] eingeführt, hier als CORBA-IDL bezeichnet. CORBA-IDL definiert ein Datentypsystem, das mit denen der Programmiersprachen C++ bzw. JAVA semantisch vergleichbar ist und Grundlage der Spezialisierungen des Konzeptes Datentyp von *CoRE$_{CEPT}$* ist. Interfacedefinitionen in CORBA-IDL beinhalten potentiell die Interaktionselemente Operation und Attribut in einer mit den Konzepten Operation und Attribut von *CoRE$_{CEPT}$* äquivalenten Weise. Für Operationen ist das Konzept Ausnahme (*Exception*) definiert. Für CORBA-IDL sind verschiedene Ableitungsregeln in Programmiersprachen (*Language Mappings*) normiert. Diese Ableitungsregeln werden auf der Seite der CORBA-Objektimplementierung benutzt, um für Operationen entsprechendes Verhalten im Kontext von CORBA-Objekten zu realisieren. Auf der Seite eines Klienten werden die Ableitungsregeln verwendet, um an Interfacedefinitionen bereitgestellte Operationen und Attribute zu benutzen. Eine umfassende Einführung in CORBA in Kombination mit C++ bietet [HV 99].

1.2.1.2 Einordnung in CoRE$_{WARE}$

CORBA stellt eine Infrastruktur bereit, die sich in idealer Weise als Basis einer *Distributed-Processing*-Umgebung eignet. Diese Infrastruktur bietet Konzepte, die eine offene, portable Lösung zur Unterstützung der operationalen Interaktion zwischen Komponenten verteilter Anwendungen enthält. Die Unterstützung der Signalinteraktion wird durch die *Object*-Services *Event*- und *Notification*-Service auf der Grundlage der durch CORBA realisierten Unterstützung operationaler Interaktionen bereitgestellt. *Continuous-Media*-Interaktionen werden mittels CORBA-Produkten gegenwärtig nicht direkt unterstützt, dennoch kann CORBA zu einer Plattform für die Steuerung multimedialer Datenströme erweitert werden (vgl. Abschnitt 1.2.2).

Die Semantik von Objekten in CORBA schließt ein, daß ein CORBA-Objekt *genau ein* Interface anbietet. Insofern ist die Idee von *CoRE*, daß COs den Kontext für mehrere genutzte bzw. angebotene Interfaces bilden, nicht direkt mittels des Konzeptes CORBA-Objekt realisierbar. Jedoch kann eine Komposition von CORBA-Objekten diesen Kontext nachbilden.

Offensichtlich eignet sich CORBA zwar als Grundlage einer *Distributed-Processing*-Umgebung, jedoch nicht direkt als *Component-Support*-Plattform. Eine *Component-Support*-Plattform kann die durch CORBA bereitgestellten Mechanismen zur Unterstützung der operationalen und Signalinteraktion nutzen. Für *Continuous-Media*-Interaktionen sind durch die *Component-Support*-Plattform auf der Basis von CORBA geeignete Mechanismen zu realisieren, die die Steuerung multimedialer Datenströme unterstützen. Eine *Component-Support*-Plattform, die auf der Basis von CORBA entsteht, muß Mechanismen realisieren, die das Konzept eines Interfacetyps als Kontext für Interaktionselemente unterschiedlicher Interaktionsarten sowie das Konzept eines CO-Typ als Kontext für mehrere angebotene bzw. genutzte Interfaces repräsentieren.

1.2.2 Continuous-Media-Delivery-Plattform

1.2.2.1 Charakterisierung

Am Lehrstuhl für Systemanalyse der Humboldt-Universität zu Berlin wurde in einem gemeinsam mit *Nippon Telegraph and Telephon Corporation* (NTT) durchgeführten Forschungs- und Entwicklungsprojekt eine verteilte Infrastruktur zur Komposition und zum Transport multimedialer Inhalte konzipiert und prototypisch realisiert [TTK+ 00][TTK+ 00a][KT99]. Diese Infrastruktur wurde auf der Basis von CORBA für unterschiedliche Technologien von Übertragungsnetzen (Internetprotokolle, Breitbandtechnologien) und unterschiedliche Arten multimedialer Inhalte (z.B. *Audio*, *Video*, *Still Picture*) und deren Kombination entworfen. Der Entwurf der *Continuous-Media-Delivery*-Plattform orientiert sich an RM-ODP und wird im folgenden skizziert.

INFORMATIONSMODELL. Im Kontext der *Continuous-Media-Delivery*-Plattform bilden die Konzepte *Content* (Inhalt) und *Medium* den Kern des entworfenen Informationsmodells (vgl. Abb.2). Das Konzept *Content* komponiert ein oder mehrere Szenen (*Scene*). Szenen wiederum sind Container für ein oder mehrere Medien (*Media*). Eine Szene beschreibt, welche Ereignisse (Zeitereignisse sowie andere Ereignisquellen) die Präsentation eines enthaltenen Mediums auslösen. Unter dem Begriff Präsentation eines Mediums wird dabei der Transport der Mediendaten von einer Quelle zu einer Senke auf der Maschine eines Konsumenten und die Darstellung der Mediendaten für einen Konsumenten verstanden. Das Konzept *Content* faßt dabei verschie-

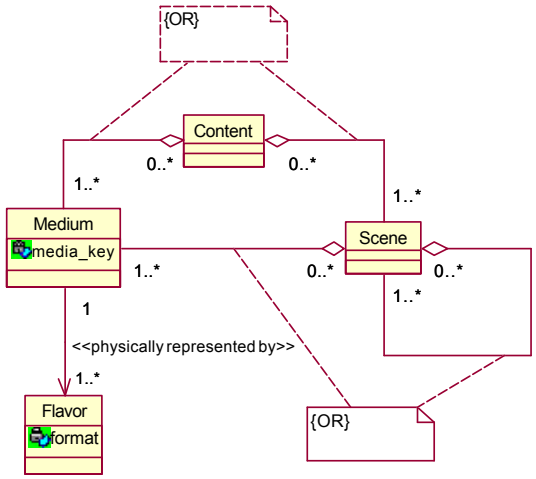


Abb. 2 Informationsmodell der Continuous-Media-Delivery-Plattform

dene Szenen zusammen, und definiert Ereignisse, die die Aktivierung einer Szene auslösen. Das Konzept Medium enthält Informationen (Meta-Daten) über nicht kombinierte, multimediale Inhalte, wie beispielsweise eine Beschreibung des Inhalts, den Autor etc., sowie Verweise auf ein oder mehrere konkret vorliegende binäre Repräsentationen (elektronische Form des multimedialen Inhaltes, als *Flavours* bezeichnet). Diese Repräsentationen umfassen zum einen die konkreten Mediendaten (z.B. eine MPEG-2-Datei oder die Referenz auf einen Echtzeit-MPEG-*Encoder*) sowie eine Beschreibung der Eigenschaften dieser Mediendaten bezüglich der Präsentationsgüte. Die Aktivierung einer Szene bedeutet die Präsentation aller enthaltenen Medien entsprechend des Eintretens von Ereignissen, die für eine Szene definiert sind.

COMPUTATIONAL-MODELL. Die *Continuous-Media*-Plattform definiert eine Menge von Rollen, die durch Softwarekomponenten repräsentiert sind, die in einer verteilten CORBA-Infrastruktur ausgeführt werden und über Interfaces interagieren. Die Komponenten der Plattform umfassen (vgl. Abb.3):

- Konsumenten (*Consumer*)

  Konsumenten nutzen multimediale Inhalte. Softwarekomponenten, die durch Konsumenten benutzt werden, realisieren den Empfang multimedialer Daten und deren Darstellung auf einem geeigneten Gerät.

- *Content*-Anbieter (*Content Provider*)

  Content-Anbieter verfügen über multimediale Daten und deren Beschreibungen (Meta-Daten). Die multimedialen Daten sowie Meta-Daten werden mit dem Ziel, an Konsumenten im Kontext von Szenen ausgeliefert zu werden, einem *Content*-Dienstanbieter angeboten.

- *Content*-Dienstanbieter (*Content Aggregator and Service Provider*)

  Content-Dienstanbieter verknüpfen Konsumenten und *Content*-Anbieter, indem sie aus angebotenen multimedialen Daten und den sie beschreibenden Meta-Daten Szenen und Inhalte komponieren und diese an Konsumenten auf deren Anforderung ausliefern. *Content*-Dienstanbieter nutzen eine durch die *Continuous-Media-Delivery*-Plattform angebotene Kommunikationssteuerung (*Communication Control*).

- Kommunikationssteuerung (*Communication Control*)

  Die Kommunikationssteuerung als zentraler Bestandteil der *Continuous-Media-Delivery*-Plattform ist für den Transport multimedialer Daten verantwortlich. Dazu werden Kommunikationskanäle von deren Quellen (i.allg. Maschinen eines *Content*-Anbieters) zu Senken (Maschinen eines Konsumenten) bereitgestellt und gesteuert. Diese Kommunikationskanäle werden durch Netzverbindungen realisiert, die durch *Network*-Dienstanbieter bereitgestellt werden.

- *Network*-Dienstanbieter (*Network Service Provider*)

  Network-Dienstanbieter steuern eine Netzinfrastruktur, die Ende-zu-Ende-Verbindungen zum Transport multimedialer Daten bereitstellt. Diese Netzinfrastruktur kann auf unterschiedlichen Technologien basieren, wobei die spezifischen Eigenschaften von konkreten Ende-zu-Ende-Verbindungen auf der Grundlage der Beschreibung der multimedialen Daten selbst zwischen der Kommunikationssteuerung und einem oder mehreren *Network*-Dienstanbietern verhandelt werden können.

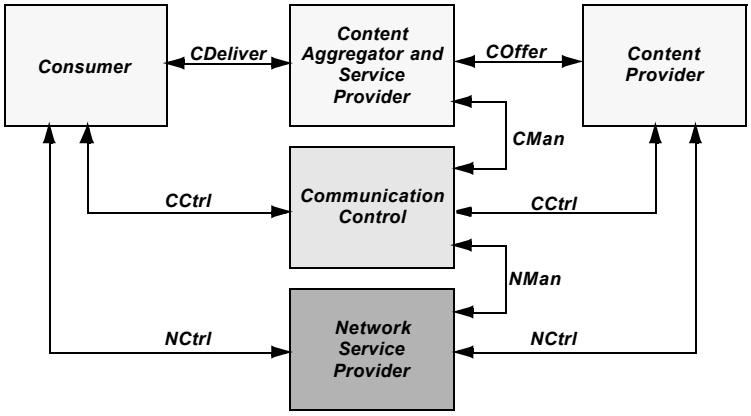


Abb. 3 Bestandteile der Continuous-Media-Plattform und ihre Interaktionspunkte

Die Zusammenfassung von Interfacedefinitionen bezüglich der potentiellen Interaktionen zwischen zwei Bestandteilen der Plattform wird als Interaktionspunkt bezeichnet. Die definierten Interaktionspunkte im Kontext der *Continuous-Media-Delivery*-Plattform sind:

- **CDeliver** zwischen Konsumenten und *Content*-Dienstanbietern,
- **COffer** zwischen *Content*-Anbietern und *Content*-Dienstanbietern,
- **CCtrl** zwischen Konsumenten und Kommunikationssteuerung bzw. zwischen *Content*-Anbietern und Kommunikationssteuerung,
- **NCtrl** zwischen Konsumenten und *Network*-Dienstanbietern sowie zwischen *Content*-Anbietern und *Network*-Dienstanbietern,
- **CMan** zwischen *Content*-Dienstanbietern und Kommunikationssteuerung sowie
- **NMan** zwischen *Network*-Dienstanbietern und Kommunikationssteuerung.

Die verwendete Infrastruktur basiert auf CORBA, dies bedeutet, daß alle Interaktionspunkte als Menge von CORBA-IDL-Interfacedefinitionen spezifiziert wurden. Die konkreten Spezifikationen sowie eine ausführliche Beschreibung der Realisierung der Gesamtlösung sind in [KKS 00] enthalten.

1.2.2.2 Einordnung in CoRE$_{WARE}$

Die Grundlage der *Continuous-Media*-Interaktionsart im Kontext von *CORE* ist - äquivalent zur Konzeption der *Continuous-Media-Delivery*-Plattform - das Konzept Medium. Ein Medium ist dabei eine Abstraktion von konkret vorliegenden binären Repräsentationen ein und desselben Multimediaobjektes gemeinsam mit den Meta-Daten dieses Objektes. Bei *Continuous-Media*-Interaktionen werden die Daten eines Repräsentanten des Mediums von einer Datenquelle zu einer Datensenke transportiert und in der Umgebung der Datensenke (d.h. auf der gleichen Maschine) präsentiert. Der Zugang zu den Daten eines Mediums erfolgt in der *Content*-Anbieter-Domäne mittels der CORBA-IDL-Interfacedefinition **MediaManager**. Diese Interfacedefinition erlaubt den Zugriff auf Meta-Informationen über ein Medium sowie auf Geräte, die die Daten eines Mediums auf der Maschine der Quelle codieren und via Ende-zu-Ende-Verbindungen transportieren können (z.B. MP3-*Encoder* in Kombination mit einem UDP-Sender). Diese Geräte werden durch die CORBA-IDL-Interfacedefinitionen **MediaDevice** und **NetworkDevice** abstrahiert. Auf der Maschine der Senke werden die gleichen Abstraktionen zum Empfang von Mediendaten und deren Darstellung (z.B. Abstraktionen von einem UDP-Empfänger in Kombination mit einem MP3-*Decoder*- und Ausgabegerät) genutzt.

Für eine Kombination der Konzepte dieser *Continuous-Media-Delivery*-Plattform mit einer *Component-Support*-Plattform sind aufgrund der konzeptionellen Verwandtschaft des Begriffes Medium in der *Continuous-Media-Delivery*-Plattform und in *CORE*$_{CEPT}$ die Abstraktionen relevant:

- **MediaManager** als Zugang zu den Meta-Daten und Daten eines Mediums,
- **MediaDevice** als Zugang zu Geräten, die Mediendaten produzieren bzw. empfangen und darstellen sowie
- **NetworkDevice** als Zugang zu den Enden von Netzverbindungen.

1.2.3 CORBA Components und Enterprise Java Beans

1.2.3.1 Charakterisierung

Im Rahmen der Standardisierung von CORBA wurden durch die *Object Management Group* zwei zunächst unabhängige RFP-Dokumente [OMG CCM RFP] [OMG MICRFP] veröffentlicht, die im Kern nach einer Komponentenarchitektur für CORBA fragen. Diese RFP-Dokumente wurden vereinigt. Im August 1999 konnte eine Lösung, an der eine Vielzahl von Organisationen beteiligt waren, in der *Object Management Group* als Technologie akzeptiert werden, die *Enterprise Java Beans* verallgemeinert.

Diese Lösung wurde als CORBA *Components* (CORBA-Komponentenmodell, [OMG CCMI]) bezeichnet und umfaßt die folgenden Schwerpunkte:

KOMPONENTENMODELL. In Erweiterung zum Meta-Typ CORBA-Objekt wurde durch das CORBA-Komponentenmodell der Meta-Typ CORBA *Component* (CORBA-Komponente) eingeführt. Der Meta-Typ CORBA *Component* definiert einen Kontext für das Anbieten bzw. Benutzen der Interfaces mehrerer CORBA-Objekte in Kombination mit der Komponentenimplementierung. Grundsätzlich kapselt eine CORBA-Komponente also die Realisierung des Anbietens bzw. Benutzens der Interfaces von CORBA-Objekten durch Implementierungen und stellt ihren Klienten eine Menge wohldefinierter Verbindungspunkte zur Verfügung (s. Abb.4).

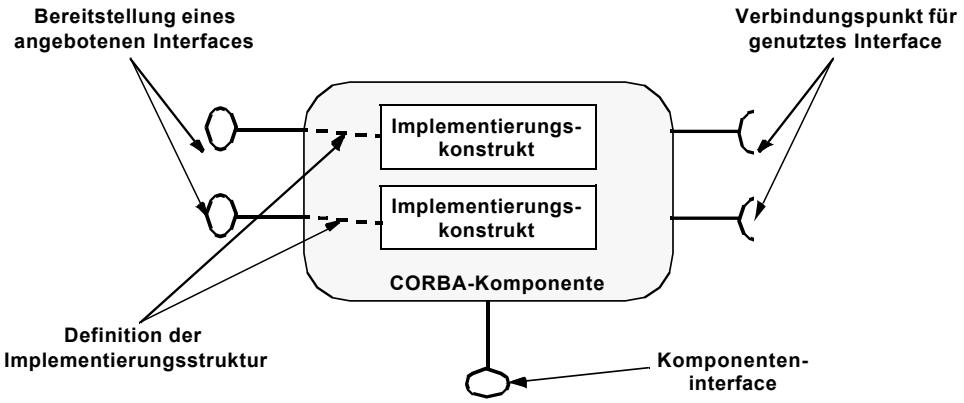


Abb. 4 CORBA-Komponente - externe und interne Sicht

Eine CORBA-Komponente definiert diese Verbindungspunkte durch das *Port*-Konzept. *Port*-Definitionen dienen - prinzipiell äquivalent zum *Port*-Konzept von *CoRE$_{CEPT}$* - der Hinterlegung bzw. der Beschaffung von CORBA-Interfacereferenzen (Referenzen von CORBA-Objekten). Im CORBA-Komponentenmodell werden *Port*-Definitionen entsprechend den Interaktionsarten operationale Interaktion und Signalinteraktion spezialisiert. Dementsprechend enthält das CORBA-Komponentenmodell:

- die *Port*-Spezialisierung *Facet* für Referenzen operationaler CORBA-Interfaces, die von einer CORBA-Komponente gegenüber ihren Klienten angeboten werden, zwischen *Facet*-Definitionen wird einem Klienten einer CORBA-Komponente Navigation bereitgestellt,
- die *Port*-Spezialisierung *Receptacle* zur Beschreibung der Möglichkeit, Referenzen namentlich unterscheidbarer CORBA-Interfaces an einer CORBA-Komponente zu hinterlegen,
- die *Port*-Spezialisierung *Event Source* als Beschreibung der Fähigkeit einer CORBA-Komponente, Signale eines spezifizierten Typs via namentlich unterscheidbaren Verbindungspunkten an interessierte Konsumenten versenden zu können,
- die *Port*-Spezialisierung *event sink* als Beschreibung eines namentlich unterscheidbaren Verbindungspunktes, an dem eine CORBA-Komponente Signale eines spezifizierten Typs empfangen kann.

Das *Port*-Konzept definiert also die Basis, auf der eine CORBA-Komponente, im Gegensatz zum Meta-Typ CORBA-Objekt, mehr als ein CORBA-Interface anbieten bzw. nutzen kann (Komposition von CORBA-Interfaces in einem gemeinsamen Kontext). Weitere, mit dem CORBA-Komponentenmodell eingeführte Konzepte beinhalten die für Klienten nutzbare Möglichkeit der Identifikation von CORBA-Komponenten (*primary key*) sowie den neuen Meta-Typ *Home*, dessen Instanzen Operationen zur Erzeugung und zum Auffinden (*Home Finder*) von CORBA-Komponenten enthalten.

Die Beschreibungssprache CORBA-IDL wurde um Konstrukte zur Definition von CORBA-Komponenten erweitert. Für diese Erweiterungen wurden keine konkreten Ableitungsregeln für Programmiersprachen, sondern Transformationsregeln zur Erzeugung von CORBA-IDL in der Version 2.4 definiert. Konkret werden CORBA-IDL-Interfacedefinitionen für eine CORBA-Komponente (*component's equivalent interface*) und CORBA-IDL-Interfacedefinitionen für alle an *Port*-Definitionen referenzierten CORBA-Interfaces erzeugt.

RAHMENWERK ZUR IMPLEMENTIERUNG VON KOMPONENTEN. Das Rahmenwerk zur Implementierung von Komponenten (*Component Implementation Framework, CIF*) definiert ein Programmiermodell zur Realisierung von CORBA-Komponenten. Die Strukturierung der Implementierung einer konkreten CORBA-Komponente wird mittels der Beschreibungssprache CIDL (*Component Implementation Definition Language*) vorgenommen, deren Sprachdefinition ebenfalls Bestandteil des Standards CORBA *Components* ist. Mit Hilfe der Beschreibungssprache kann die Struktur programmiersprachlicher Konstrukte definiert werden, die das Verhalten einer CORBA-Komponente realisieren. Eine Strukturierung (bezeichnet als Segmentierung) der Implementierung einer CORBA-Komponente ist mittels CIDL möglich, jedoch ist der Feinheitsgrad der Segmentierung auf die Implementierung mindestens einer CORBA-IDL-Interfacedefinition beschränkt[1]. CIDL enthält des weiteren eine Integration mit dem *Persistent-State*-Service (PSS, [OMG PSS 2.0]), der es im Kontext der Implementierungsbeschreibung für eine CORBA-Komponente erlaubt, die Nutzung persistenter Speichertechnologien für konkrete Zustandsattribute einer CORBA-Komponente zu definieren. Für die Beschreibungssprache CIDL wurden im Rahmen der *Object Management Group* bisher keine Ableitungsregeln für konkrete Programmiersprachen vorgelegt, so daß damit zum gegenwärtigen Zeitpunkt das gesamte Rahmenwerk zur Implementierung von Komponenten nicht genutzt werden kann. Der hier präsentierte Ansatz der automatischen Ableitung von Softwarekomponenten kann zur Definition von Ableitungsregeln für CIDL-Spezifikationen herangezogen werden, da er hinsichtlich seiner unterstützten Konzepte weit über CIDL hinausgeht.

AUSFÜHRUNGSUMGEBUNG FÜR KOMPONENTEN. Als Ausführungsumgebung für CORBA-Komponenten definiert das CORBA-Komponentenmodell das *Container*-Programmiermodell (*Container Programming Model*). Ein *Container* ist die serverseitige Ausführungsumgebung für eine CORBA-Komponentenimplementierung, eine Menge von Containern formt konzeptionell eine *Component-Support*-Plattform. Das *Container*-Programmiermodell umfaßt die folgenden Programmierinterfaces (*Application Programming Interface, API*):

- externe Programmierinterfaces (*External API Types*) sind definiert durch CORBA-IDL-Interfaces, die dem Klienten einer CORBA-Komponente zur Verfügung stehen,
- *Container*-Programmierinterfaces (*Container API Types*) enthalten Interfaces, die einem Entwickler bei der Implementierung einer CORBA-Komponente zur Verfügung stehen,
- das CORBA-Nutzungsmodell (CORBA *Usage Model*) spezifiziert die Interaktionen zwischen einem Container und der dem Container zugrundeliegenden *Distributed-Processing*-Umgebung,
- die Programmierinterfaces der Komponentenkategorie, die die Kombination der externen Programmierinterfaces mit den *Container*-Programmierinterfaces definieren.

RAHMENWERK ZUR *DEPLOYMENT*- UND INSTALLATIONSUNTERSTÜTZUNG. Die Definition von *Deployment*-Vorgängen für Implementierungen von CORBA-Komponenten ist ebenfalls Bestandteil des CORBA-Komponentenmodells. Die Grundlage von *Deployment*-Vorgängen bilden Softwarepakete, die eine oder mehrere Implementierungen einer oder mehrerer CORBA-Komponenten enthält. Diese Softwarepakete können entweder singulär auf Maschinen installiert, oder aber mit Implementierungen weiterer CORBA-Komponenten gruppiert werden. Eine solche Gruppierung wird in CORBA *Components* als *Assembly* bezeichnet. Software-

1. Im Kontext von *CoRE* entspricht das Prinzip der Segmentierung der Implementierung einer CORBA-Komponente dem Konzept der Realisierung von Interaktionselementen durch Implementierungselemente eines Artefakts, jedoch ist diese Strukturierung wesentlich feinkörniger.

pakete enthalten neben den in Dateien vorliegenden Implementierungen einer CORBA-Komponente weitere Dateien, die die Eigenschaften und Anforderungen des Softwarepaketes beschreiben. Das Rahmenwerk zur *Deployment*- und Installationsunterstützung, insbesondere die Struktur der Beschreibungen von Softwarepaketen basieren auf [W3COSD].

1.2.3.2 Einordnung in CoRE$_{WARE}$

Das CORBA-Komponentenmodell bietet eine Realisierung einer *Component-Support*-Plattform. Es enthält eine einheitliche Sicht auf Softwarekomponenten und deren Ausführung. Das Konfigurationsmanagement wird durch die Einführung des *Port*-Konzeptes unterstützt. Eine spezifische, simplifizierte Ausprägung des CORBA-Komponentenmodells beinhaltet *Enterprise Java Beans* (EJB, [SunEJB]), so daß eine Betrachtung der Verallgemeinerung von EJB im CORBA-Komponentenmodell hinreichend ist, um beide Ansätze bewerten zu können.

Der wesentliche Schwachpunkt besteht in der Unvollständigkeit des Standards, insbesondere in der Nichtverfügbarkeit von Programmiersprachenabbildungen der Beschreibungssprache CIDL. Diese Unvollständigkeit sowie die widersprüchlichen, ebenfalls unvollständigen Spezifikationen der anderen Bestandteile hatten eine geringe Akzeptanz des Standards sowohl im industriellen als auch im akademischen Umfeld zur Folge. In Kapitel 3 von [CoRE I] wurde bereits diskutiert, daß die Konzepte des CORBA-Komponentenmodells eine zentrale semantische Basis von *CoRE* bilden, konkret werden diese Konzepte durch *CoRE$_{CEPT}$* erweitert - die Realisierung der *Component-Support*-Plattform *CoRE$_{WARE}$* und der Ableitungsregeln *CoRE$_{MAP}$* stellt also eine praktische Umsetzung des CORBA-Komponentenmodells dar.

1.2.4 Component Object Model

1.2.4.1 Charakterisierung

Component Object Model (*COM*, [Box 99]) wurde durch die Firma *Microsoft Corp.* im Jahre 1993 eingeführt und wird seitdem in den Betriebssystemen dieses Unternehmens als Komponentenarchitektur eingesetzt. COM definiert einen programmiersprachenunabhängigen, objektorientierten, binären Interoperabilitätsansatz als Infrastruktur für die Integration von Softwarekomponenten. COM erlaubt die Nutzung der durch eine Softwarekomponente bereitgestellten Funktionalität unabhängig von der zur Realisierung dieser Komponente verwendeten Programmiersprache.

COM nutzt die Termini Objekt, Klasse, Interface und Komponente. Ein Objekt im Kontext von COM ist eine konkrete Instanz einer namentlich identifizierbaren Klasse, eine Klasse ist hier die Implementierung eines oder mehrerer Interfaces. Jedes Interface repräsentiert in COM eine Gruppierung von in logischer Relation stehenden Funktionen, als Methoden bezeichnet. Ein Interface bezeichnet somit einen streng getypten Kontrakt zwischen dem Nutzer eines Objektes und dem Objekt selbst, das dieses Interface anbietet. COM definiert im Gegensatz zu CORBA kein über Maschinengrenzen hinaus gültiges Referenzkonzept. Anstelle dessen erhält man den Zugang zu Interfaces eines Objektes über das Konzept *Interface Pointer*. Jedes COM-Objekt unterstützt mindestens das Interface **IUnknown**, alle weiteren Interfaces entstehen durch Spezialisierung von **IUnknown**. In COM wird die Schnittstellenbeschreibungssprache *Microsoft Interface Definition Language (MIDL)* genutzt, um konkrete Interfaces mit deren Methoden zu beschreiben. MIDL ist eine objektorientierte Erweiterung der durch *Open Software Foundation* (*OSF*) definierten Sprache *Interface Definition Language*, die im Kontext von *Distributed Computing Environment* (*DCE*) entwickelt wurde. Diese Erweiterungen hatten zum Ziel, Objektorientierung im Kontext verteilter Softwaresysteme zu unterstützen.

1.2.4.2 Einordnung in CoRE$_{WARE}$

Durch die Integration von COM in das weit verbreitete Betriebssystem *Windows* der Firma Microsoft hat sich dieser Ansatz als Komponentenarchitektur zur Integration von Softwarekomponenten auf einer

Maschine durchgesetzt. Die Softwarekomponenten von Anwendungen, die auf einer Maschine mit dem Betriebssystem *Windows* ausgeführt werden, interagieren miteinander und mit den Komponenten anderer Anwendungen auf der gleichen Maschine unter Nutzung von COM. Die Definition von *Distributed COM* (DCOM) gilt als Versuch der Firma *Microsoft*, die Nutzung von COM auf die Interaktion zwischen Komponenten auf verschiedenen Maschinen auszudehnen. DCOM konnte sich jedoch bisher nicht als Komponentenarchitektur für verteilte Softwaresysteme - insbesondere im Telekommunikationskontext - durchsetzen, wie durch einer durch das Unternehmen *Sun Microsystems* durchgeführten und veröffentlichen Studie mit dem Titel "*Industry Opinions On Enterprise Java Beans (EJB) vs. COM+/MTS*" belegt wurde [SunEJBa]. Insofern betrachten der Autor die Komponentenarchitekturen COM und DCOM als nicht praktikabel für Integration und Einsatz von Softwarekomponenten verteilter Telekommunikationssoftwaresysteme.

Microsoft hat mit der Ankündigung der als *.net* bezeichneten Architektur die extensive Nutzung von *Internet*-Standards als Grundlage einer neu zu definierenden Komponentenarchitektur erklärt. Eine derartige Strategie würde sicherlich, sofern sie umgesetzt werden kann, eine größere Akzeptanz dieser Komponentenarchitektur für verteilte Softwaresysteme bewirken. Jedoch sind zum gegenwärtigen Zeitpunkt weder die Architektur *.net* noch eine dazugehörige Komponentenarchitektur fertiggestellt, so daß deren technologische Grundlagen im Rahmen von $CORE_{WARE}$ nicht berücksichtigt werden konnten.

1.2.5 Fazit

Sowohl die *Continuous-Media-Delivery*-Plattform als auch CORBA *Components* nutzen CORBA als Basis der zugrundeliegenden *Distributed-Processing*-Umgebungen.

Die *Continuous-Media-Delivery*-Plattform definiert kein Komponentenmodell, sondern eine verteilte Infrastruktur für multimediale Daten, deren Übertragung durch Einsatz verschiedener Technologien von einer Datenquelle zu einer Datensenke in der Domäne eines Konsumenten realisiert wird. Die Konzepte dieser Plattform eignen sich als Ergänzung einer *Distributed-Processing*-Umgebung um eine Unterstützung für *Continuous-Media*-Interaktionen.

Im Gegensatz dazu definiert das CORBA-Komponentenmodell eine *Component-Support*-Plattform. Das Modell unterstützt die Nutzung bzw. Bereitstellung mehrerer CORBA-Interfaces im Kontext einer CORBA-Komponente durch die Einführung von *Port*-Definitionen. *Port*-Definitionen werden für CORBA-Interfacedefinitionen mit verschiedenen Interaktionsarten unterschieden, wobei das CORBA-Komponentenmodell ausschließlich die operationale und Signalinteraktion berücksichtigt. Eine CORBA-Komponente enthält ihre Implementierung, die mittels der Beschreibungssprache CIDL strukturiert werden kann. Insofern bietet CIDL und das Rahmenwerk zur Komponenten-Implementierung verglichen mit der Implementierungssicht von *CORE* einen prinzipiell äquivalenten Ansatz. Sowohl CIF als auch die Implementierungssicht unterstützen das Prinzip der Strukturierung einer Implementierung einer CORBA-Komponente bzw. des semantisch äquivalenten Konzeptes CO-Typ. Jedoch unterscheiden sich beide Ansätze bezüglich der Flexibilität dieser Beschreibungen: Während mittels CIF eine Segmentierung der Implementierung basierend auf den unterstützten CORBA-IDL-Interfacedefinitionen erreicht werden kann, bietet die Implementierungssicht von *CORE* eine feinkörnigere Segmentierung der Implementierung basierend auf den in Interfacetypen enthaltenen Interaktionselementen.

Das CORBA-Komponentenmodell unterstützt die operationale und Signalinteraktion, *CORE* erweitert die Menge der Interaktionsarten im CORBA-Komponentenmodell um *Continuous-Media*-Interaktion. Eine Kombination der Prinzipien des CORBA-Komponentenmodells mit der Erweiterung der zugrundeliegenden *Distributed-Processing*-Umgebung um die Ansätze der *Continuous-Media-Delivery*-Plattform liegt zur Unterstützung von *Continuous-Media*-Interaktionen nahe. Dabei wird eine Integration des Konzeptes Medium von $CORE_{CEPT}$ mit dem Konzept Medium der *Continuous-Media-Delivery*-Plattform angestrebt.

1.3 Component-Support-Plattform von CoRE

Basierend auf den Konzepten der vorgestellten Plattformtechnologien, insbesondere des Standards CORBA *Components* - wurde die *Component-Support*-Plattform *CoRE$_{WARE}$* entworfen, deren Mechanismen eine Abbildung aller Konzepte von *CoRE$_{CEPT}$* erlauben. Diese Ableitungsregeln - *CoRE$_{MAP}$* - werden in Kapitel2 dargestellt und diskutiert. Damit sind die Voraussetzungen für die Entwicklung von Entwicklungswerkzeugen gegeben, die die Anwendung dieser Regeln auf konkrete Entwurfsmodelle automatisieren.

COs interagieren mit ihrer Umgebung über genutzte oder angebotene Interfaces. Dem in *CoRE$_{CEPT}$* definierten Konzeptraum entsprechend kombinieren die Typen dieser Interfaces Interaktionselemente der drei Interaktionsarten Signalinteraktion, *Continuous-Media*-Interaktion und operationale Interaktion. COs stellen den gemeinsamen Kontext für die Menge ihrer angebotenen bzw. genutzten Interfaces her. *CoRE$_{WARE}$* definiert eine plattformspezifische Repräsentation von CO-Typen. Diese Repräsentation enthält zum einen eine *externe Sicht* bezüglich eines CO-Typs, zum anderen wird die in einem konkreten Entwurfsmodell definierte Implementierungsstruktur als *interne Sicht* bezüglich eines CO-Typs bezeichnet.

Neben der Repräsentation der CO-Typen ermöglicht *CoRE$_{WARE}$* die transparente Integration der von der zu Grunde liegenden *Distributed-Processing*-Umgebung bereitgestellten Mechanismen zur Umsetzung der unterschiedlichen Interaktionsarten. Diese Mechanismen sind zu einem Teil spezifisch für jeden CO-Typ, beispielsweise die Steuerung von entkoppelten Kommunikationskanälen der *Distributed-Processing*-Umgebung für die Realisierung von Signalinteraktion, und werden durch technologiespezifische Entwicklungswerkzeuge bereitgestellt. Andererseits sind allgemeine Mechanismen identifizierbar, die durch *CoRE$_{WARE}$* direkt bereitgestellt werden. Zu diesen Mechanismen gehört die für einen Entwickler transparente Verwaltung von Zustandsattributen für CO-Typen. Sowohl diese spezifischen als auch die allgemeinen Mechanismen bilden die *Ausführungsumgebung* für COs.

Während die externe Sicht zwar abhängig von der gewählten *Distributed-Processing*-Umgebung, aber unabhängig von der gewählten Implementierungstechnologie ist, läßt sich die interne Sicht nur in Abhängigkeit von der gewählten Implementierungstechnologie beschreiben - im Falle von *CoRE$_{WARE}$* die Komponentenarchitektur CORBA sowie die Programmiersprache C++ [ANSI C++].

Die interne Sicht auf einen CO-Typ ist charakterisiert durch die das Verhalten des CO-Typs realisierenden programmiersprachlichen Konstrukte, abstrahiert durch das Konzept Artefakt. Die internen Aspekte eines CO-Typs sind beschrieben durch die Zuordnung von Interaktionselementen zu programmiersprachlichen Konstrukten im Kontext der durch Artefakte beschriebenen Konstrukte. Diese Zuordnung ist im Entwurfsmodell durch Implementierungselemente spezifiziert. *CoRE$_{MAP}$* realisiert die transparente Anbindung der durch die *Distributed-Processing*-Umgebung bereitgestellten Objektinteraktionsmechanismen an die interne Konfiguration.

Instanziierungsmuster für Artefakte können eine dynamische Instanziierung der die Artefakt realisierenden programmiersprachlichen Konstrukte erforderlich machen. Beispielsweise werden bei der Instanziierungsregel ***ARTIFACT_PER_REQUEST*** während jeder Interaktion mit dem CO neue Instanzen der programmiersprachlichen Konstrukte erzeugt und nach Beendigung der Interaktion wieder zerstört. Um die programmiersprachlichen Konstrukte erzeugen zu können, werden korrespondierende Fabriken (*Artifact Factories*) bereitgestellt.

Für die externe Sicht stellt die *Component-Support*-Plattform Ausdrucksmittel bereit, die - basierend auf den Konzepten der genutzten *Distributed-Processing*-Umgebung - den Zugriff auf das CO über die definierten Interfaces erlauben. Dabei werden

- die in einem konkreten Entwurfsmodell spezifizierten Relationen zwischen Interfacetypen eines CO-Typs und dem CO-Typ selbst erhalten,
- die Zugehörigkeiten von Interaktionselementen potentiell verschiedener Interaktionsarten zu ein und demselben Interfacetyp repräsentiert sowie

- Mechanismen zum Auffinden bzw. Erzeugen von COs bereitgestellt.

Die Relation zwischen Interfaces eines COs und dem CO selbst widerspiegelt sich jedenfalls in Restriktionen bezüglich der Lebenszeit von Konstrukten der *Distributed-Processing*-Umgebung, die die Interfacetypen bzw. den CO-Typ selbst realisieren. Es wird insbesondere sichergestellt, daß die Lebenszeit der Konstrukte, die die Interfacetypen eines CO-Typs implementieren, auf die Lebenszeit der Konstrukte, die den CO-Typ selbst implementieren, beschränkt ist. Darüber hinaus stellt $CoRE_{WARE}$ Mechanismen bereit, die die Relation zwischen Interfaces eines COs und dem CO selbst für die Umgebung des COs weitergehend manifestieren. Dazu gehören

- die Möglichkeit, von einem, ein Interface eines COs repräsentierenden Konstrukt, auf das das CO selbst realisierende Konstrukt zu schließen (*CO-Navigation*),
- in der entgegengesetzten Richtung von einem CO selbst auf alle Konstrukte zu schließen, die die Interfaces dieses COs repräsentieren (*Interfacenavigation*),
- Mechanismen, die den Typ eines CO mit seinen *Port*-Definitionen beschreiben (*Reflexion*).

Die Repräsentation von Interaktionselementen potentiell unterschiedlicher Interaktionsarten in einer konkreten *Distributed-Processing*-Umgebung ist selbstverständlich von der der *Distributed-Processing*-Umgebung zu Grunde liegenden Basistechnologie abhängig. In einer CORBA-basierten *Distributed-Processing*-Umgebung wird beispielsweise die Signalinteraktion durch *Event* bzw. *Notification*-Kanäle realisiert, während operationale Interaktion bereits durch CORBA selbst ermöglicht wird. Die *Component-Support*-Plattform integriert bei der Abbildung der Typen der Interfaces eines CO-Typs diese Technologieabhängigkeiten transparent: Diese Interfacetypen werden als Ganzheit repräsentiert.

Die externe Sicht auf einen CO-Typ wird durch Mechanismen zur Erzeugung und zum Auffinden von COs komplettiert. Erzeugung ist der definierte Aufbau der internen Konfiguration eines COs durch Instanziierung der das CO realisierenden Konstrukte der *Distributed-Processing*-Umgebung und der dazugehörigen programmiersprachlichen Artefakte. Die Initiierung der Erzeugung eines COs kann durch die Umgebung des COs erfolgen. Die dazu notwendigen CO-Fabriken (*CO-Factories*) werden durch $CoRE_{WARE}$ und $CoRE_{MAP}$ bereitgestellt. Weiterhin wird auch das Auffinden einer geeigneten CO-Fabrik durch die *Component-Support*-Plattform unterstützt (*CO-Factory Finder*).

$CoRE_{CEPT}$ separiert die Identität eines COs von der Identität der zugehörigen Artefakte. Dieses Prinzip impliziert, daß die Konstrukte der internen Sicht ein und desselben COs zu verschiedenen Zeitpunkten unterschiedlich sein können. Im Falle der Repräsentation von Artefakten durch C++-Klassen und dem Instanziierungsmuster ***ARTIFACT_POOL*** kann ein Aufruf an einem unterstützten Interface ein und desselben COs durch unterschiedliche Instanzen (unterschiedliche Identität) dieser C++-Klassen realisiert werden. Auffinden eines COs zu einem Zeitpunkt bedeutet, den Zugriff auf die aktuelle interne Sicht bereitzustellen. Entsprechende Funktionalität wird ebenfalls durch CO-Fabriken unterstützt.

$CoRE_{WARE}$ integriert ein auf CORBA *Components* basierendes Komponentenmodell mit den Konzepten der *Continuous-Media-Delivery*-Plattform. Diese Integration wurde so vorgenommen, daß die *Delivery*-Unterstützung für Medien, wie sie durch die *Continuous-Media-Delivery*-Plattform definiert ist, integraler Bestandteil der *Distributed-Processing*-Umgebung von $CoRE_{WARE}$ ist. Somit können in Entwurfsmodellen definierte *Continuous-Media*-Interaktionselemente direkt auf die Mechanismen der *Continuous-Media*-Plattform abgebildet werden.

Auf die gleiche Weise werden Signalinteraktionselemente in Entwurfsmodellen auf CORBA-*Event*- bzw. *Notification*-Servicekanäle abgebildet. Beide Services sind Bestandteil der durch CORBA definierten *Object*-Services und dienen der asynchronen, entkoppelten Kommunikation zwischen verteilten CORBA-Objekten, die auch die konzeptionelle Basis der Signalinteraktionselemente in *CoRE* darstellt. Insofern werden Signalinteraktionselemente eines Entwurfsmodells auf durch den CORBA-*Event*- bzw. *Notification*-Service spezifizierte CORBA-IDL-Interfacedefinitionen und deren Implementierungen abgebildet.

KAPITEL 2

Ableitungsregeln für Softwarekomponenten

Entsprechend der in Abschnitt 1.3 vorgestellten Konzeption von $CoRE_{WARE}$ werden im folgenden die Ableitungsregeln für die Konzepte des Konzeptraumes konkretisiert sowie die unterstützende Ausführungsumgebung von $CoRE_{WARE}$ im Detail vorgestellt. Für alle präsentierten Ableitungsregeln werden Beispiele angegeben, die die Regelanwendung bzw. die Kombination mehrerer Regeln illustrieren.

$CoRE_{WARE}$ basiert auf CORBA 2.4 [OMG CORBA]. Für die Definition der Ableitungsregeln wird die Programmiersprache C++ [ANSI C++] exemplarisch benutzt[1].

2.1 Abbildung der Konzepte der Struktursicht

Konzepte der Struktursicht sind primär der externen Sicht auf CO-Typen zuzuordnen. Begründet in der Wahl von CORBA 2.4 als Basistechnologie der *Distributed-Processing*-Umgebung werden also durch $CoRE_{MAP}$ Definitionen der Struktursicht eines konkreten Entwurfsmodells durch CORBA-IDL-Definitionen repräsentiert.

2.1.1 Konzept Namensraum

Die Elemente eines Entwurfsmodells werden in Namensräumen definiert, sie sind entweder Bestandteil des globalen Namensraumes oder von Namensräumen im globalen Namensraum. Namensräume können in Entwurfsmodellen verschachtelt sein. Mit Hilfe von Ableitungsregeln werden die Namensräume eines Entwurfsmodells auf CORBA-IDL-Module abgebildet. Die resultierenden CORBA-IDL-Moduldefinitionen sind die Grundlage zur Definition von Namensräumen in der zur Implementierung der internen Sicht auf CO-Typen verwendeten Programmiersprache.

1. Aufgrund der Ähnlichkeiten lassen sich die Ableitungsregeln ebenso für mindestens JAVA und C# definieren.

(Regel 1) Eine Instanz des Konzeptes Namensraum wird auf eine CORBA-IDL-Moduldefinition mit dem gleichen Namen wie der der Namensrauminstanz abgebildet. Verschachtelte Instanzen von Namensraum werden durch verschachtelte CORBA-IDL-Module repräsentiert. Die Definitionen jeder Namensrauminstanz werden durch äquivalente CORBA-IDL-Definitionen in den entsprechenden CORBA-IDL-Modulen repräsentiert. Für den globalen Namensraum wird kein CORBA-IDL-Modul erzeugt.

Für die in einem Namensraum eines Entwurfsmodells definierten Modellelemente, für die CORBA-IDL-Definitionen erzeugt werden, sind diese in der CORBA-IDL-Moduldefinition vorgenommen, die durch Anwendung von *Regel 1* erzeugt wurde.

(Beispiel 1) Wenn ***M2*** als Namensraum in einem Namensraum ***M1*** eines Entwurfsmodells definiert ist, so ergibt sich die folgende CORBA-IDL-Definition:

```
module M1 {
  module M2 {
    // Definitionen des Namensraums M2
  };
  // Definitionen des Namensraums M1
};
```

2.1.2 Datentyp

Dem in [CoRE I], Kapitel3 präsentierten Konzeptraum liegt das CORBA-IDL-Datentypsystem zugrunde. Für dieses Datentypsystem definiert [OMG CCMII] ein Metamodell, das integraler Bestandteil von *CoRE*$_{CEPT}$ ist (vgl. Abschnitt 3.2.2 in [CoRE II]). Das UML-Profil für CORBA [OMG CORBA P] definiert konkrete Ableitungsregeln zur Repräsentation von entsprechend [OMG CCM II] formulierten Datentypdefinitionen in Entwurfsmodellen mittels UML. Diese Ableitungsregeln für Datentypdefinitionen sind Bestandteil von *CoRE*$_{TATIONS}$ (vgl. Abschnitt 5.2.5.2 in [CoRE II]). Obwohl diese Regeln mit der Intension definiert wurden, CORBA-IDL-Sprachkonstrukte mittels UML darzustellen, implizieren sie auch eine kanonische Transformation der Metamodellelmente von [OMG CCM II] nach CORBA-IDL. Insofern ist die Abbildung der in einem *CoRE*-basierten Entwurfsmodell verwendeten Datentypen definiert und wird hier nicht im Detail diskutiert.

(Regel 2) Instanzen der Spezialisierungen des Konzeptes Datentyp werden kanonisch auf CORBA-IDL-Definitionen abgebildet.

Mit Hilfe dieser Regel lassen sich nun sowohl primitive als auch strukturierte Datentypdefinitionen eines Entwurfsmodells in CORBA-IDL darstellen, die Anwendung der Regel ist beschränkt auf alle primitiven Datentypen, die strukturierten Datentypen **Struct** und **Valuetype** sowie auf *Template*-Definitionen (**Sequence**, **Array**) über diese Datentypen.

(Beispiel 2) Wenn ***S*** eine Strukturdefinition im Modul ***M1*** mit zwei Elementen ***m1*** und ***m2*** ist, wobei ***m1*** vom Typ ***string*** und ***m2*** vom Typ ***long*** sind, wird folgende CORBA-IDL-Definition erzeugt:

```
module M1
{
  struct S
  {
    string m1;
    long m2;
  };
};
```

2.1.3 Ausnahme

Das Konzept Ausnahme entspricht konzeptionell dem CORBA-IDL-Konstrukt *Exception*. Demzufolge werden für in einem Entwurfsmodell enthaltene Ausnahmendefinitionen CORBA-IDL-*Exceptions* erzeugt. Elemente der Ausnahmedefiniton werden zu Elementen der CORBA-IDL-*Exception*-Definition. Die konkreten Ableitungsregeln ergeben sich wiederum unter Verwendung von [OMGCORBA P].

(Regel 3) Instanzen des Konzeptes Ausnahme werden mit Hilfe der aus [OMGCORBAP] implizierten Regeln auf CORBA-IDL-*Exception*-Definitionen abgebildet.

Das potentielle Auslösen einer Ausnahme durch eine Operation im Kontext eines Interfacetyps kann durch eine CORBA-IDL-*Raises*-Definition für die CORBA-IDL-Repräsentation der Ausnahmedefinition dargestellt werden.

(Beispiel 3) Wenn ***E*** eine Ausnahme im Modul ***M1*** mit einem Element ***el1*** ist, wobei ***el1*** vom Typ ***S*** ist, wird folgende CORBA-IDL-Definition erzeugt:

```
module M1 {
  exception E {
    S el1;
  };
};
```

2.1.4 Signaltyp und Signalparameter

Der Typ eines Signalparameters ist Instanz der Datentypspezialisierung *Value*-Typ (vgl. Abschnitt 3.1.10 in [CoRE I]). Dementsprechend werden die Ableitungsregeln für das Konzept Datentyp auf Definitionen von Signalparametern in einem Entwurfsmodell angewendet, resultierend in CORBA-IDL-Konstrukten des Typs **valuetype**.

(Regel 4) Instanzen des Konzeptes Signalparameter im Entwurfsmodell werden unter Anwendung von *Regel 2* auf CORBA-IDL-**valuetype**-Definitionen abgebildet.

Signaltypdefinitionen in Entwurfsmodellen werden ebenfalls auf das CORBA-IDL-Konstrukt **valuetype** abgebildet, wobei die Signalparameter zu *Member*-Elementen dieses **valuetype**-Konstrukts werden. Die Namen dieser Elemente ergeben sich kanonisch aus den Namen der Signalparameter im Entwurfsmodell.

(Regel 5) Instanzen von Signaltyp werden auf CORBA-IDL-**valuetype**-Definitionen abgebildet. Der Name des erzeugten **valuetype**-Konstrukts ist der Name des Signaltyps. Für jeden in einer Signaltypdefinition enthaltenen Signalparameter wird eine *Public-Member*-Definition im resultierenden CORBA-IDL-**valuetype**-Konstrukt angelegt. Der Typ dieses Elements ist derjenige CORBA-IDL-**valuetype**-Typ, der für den Signalparameter angelegt wurde. Der Name des *Public-Member*-Elements entspricht dem Namen des Signalparameter des Signaltyps.

Zur Vereinfachung der Instanziierung von Signalen eines Signaltyps durch Signalfabriken (*Signal Factories*) erhalten die resultierenden **valuetype**-Konstrukte CORBA-IDL-**factory**-Operationen, mit deren Hilfe zur Ausführungszeit Instanzen dieser Signaltypen erzeugt werden können. Zur Erzeugung dieser Instanzen werden der entsprechenden **factory**-Operation die aktuellen Signalparameter als Parameter übergeben.

(Regel 6) Für jede Instanz von Signaltyp wird in dem aus *Regel 5* resultierenden CORBA-IDL-**valuetype**-Konstrukt eine **factory**-Operation mit dem Namen **create_<Signaltypname>** erzeugt. Für jede Definition eines Signalparameters wird in der **factory**-Operation ein Parameter produziert.

Instanzen von Signaltyp können in einem Entwurfsmodell in einer Vererbungsrelation stehen, dies wird durch äquivalente Vererbung der **valuetype**-Konstrukte in CORBA-IDL nachgebildet. Grundsätzlich erbt

jede Repräsentation einer Instanz von Signaltyp von einem durch die Ausführungsumgebung vorgegebenen Basiskonstrukt.

(Regel 7) Vererbungsrelationen von Instanzen des Konzeptes Signaltyp werden auf Vererbungsrelationen derjenigen CORBA-IDL-**valuetype**-Definitionen abgebildet, die für die Basissignaltypen eines Signaltyps erzeugt wurden. Repräsentationen von Signaltypen, für die keine Basissignaltypen in einem Entwurfsmodell definiert sind, erben von dem durch die Ausführungsumgebung vordefinierten CORBA-IDL-**valuetype**-Konstrukt **ComponentModel::EventBase**.

Diese Ableitungsregeln erlauben die Zusicherung des typsicheren Empfangs bzw. des typsicheren Sendens von Signalen durch $CoRE_{WARE}$: In einer konkreten Implementierung kann mit denjenigen programmiersprachlichen Konstrukten gearbeitet werden, die für die CORBA-IDL-**valuetype**-Konstrukte durch ein CORBA-IDL-*Compiler*-Werkzeug generiert werden. Dies ist sogar unabhängig vom konkreten Mechanismus zur Übertragung von Signalen in Kommunikationskanälen, wie z.B. Nutzung des CORBA-IDL-Datentyps **any** innerhalb des CORBA-*Event*-Service [OMG CORBAES] oder Nutzung der CORBA-*Notification*-Servicedefinition **StructuredEvent** [OMGCORBANoS]. Darüber hinaus wird polymorpher Signalempfang gestattet. Falls der Sender eines Signals ein Signal eines weiter spezialisierten Signaltyps versendet als der Empfänger erwartet, so kann der Empfänger diejenigen Informationen einem Signal entnehmen, die für den Signaltyp des erwarteten Basissignals definiert sind.

(Beispiel 4) Sei ***V*** eine *Value*-Typ-Definition in einem Entwurfsmodell im Namensraum ***M1***, das die in *Beispiel 2* definierte Struktur ***S*** als Element mit dem Namen ***m1*** beinhaltet. Sei weiterhin ***Sig1*** eine Signaltypdefinition im Entwurfsmodell im Namensraum ***M1***, die einen Wert des Namens ***carry_v*** vom Typ ***V*** transportiert, wobei ***Sig1*** in keiner Vererbungsrelation steht. Dann werden die folgende CORBA-IDL-Definitionen produziert:

```
module M1 {
  valuetype V {
    public S m1;
  };
  valuetype Sig1 : ComponentModel::EventBase {
    public V carry_v;
    factory create_Sig1 ( in V carry_v );
  };
};
```

Die Definition der **factory**-Operation umfaßt bei spezialisierten Signaltypen ebenfalls die *Member*-Elemente der Basissignaltypen als Konstruktionsparameter, sichergestellt durch die folgende Regel.

(Regel 8) Für Signaltypen, die in einer Vererbungsrelation stehen, werden der **factory**-Operation des für den spezialisierten Signaltyp produzierten **valuetype**-Konstrukts weitere Konstruktionsparameter übergeben, die den Typ derjenigen **valuetype**-Konstrukte haben, die für die Parameter der Basissignaltypen erzeugt wurden. Die Namen dieser Parameter entsprechen den Namen der Signalparameter, die für die Basissignaltypen im Entwurfsmodell definiert sind. In der Reihenfolge der Konstruktionsparameter erscheinen zuerst die Signalparameter des Basissignaltyps und anschließend diejenigen des spezialisierten Signaltyps.

Die Festlegung der Reihenfolge der Konstruktionsparameter ist natürlich willkürlich, muß jedoch definiert sein, um einem Entwickler, der in der Artefaktrealisierung den *Factory*-Mechanismus für Signale nutzt, vorzugeben, in welcher Reihenfolge die zu übertragenden Werte bei der Konstruktion des Signals eines Signaltyps anzugeben sind. Dies ist insbesondere dann essentiell, wenn sowohl für einen spezialisierten Signaltyp als auch dessen Basissignaltyp Signalparameter desselben Typs definiert sind. Diese Situation verdeutlicht das folgende Beispiel.

(Beispiel 5) Sei ***Sig2*** eine Signaltypdefinition im Namensraum ***M1*** derart, daß ***Sig2*** den in *Beispiel 4* definierten Signaltyp ***Sig1*** spezialisiert. Weiterhin sei für ***Sig2*** ein Signalparameter des Namens ***carry_vv*** mit dem Typ ***V*** definiert. Dann werden die folgenden CORBA-IDL-Definitionen generiert:

```
module M1 {
  valuetype V {
    public S m1;
  };

  valuetype Sig1 : ComponentModel::EventBase {
    public V carry_v;
    factory create_Sig1 ( in V carry_v );
  };

  valuetype Sig2 : Sig1 {
    public V carry_vv;
    factory create_Sig2 ( in V carry_vv, in V carry_v );
  };
};
```

2.1.5 Interfacetyp und Interaktionselement

Generell werden Instanzen des Konzeptes Interfacetyp auf CORBA-IDL-Interfacedefinitionen abgebildet. Während in Entwurfsmodellen jedoch *alle* Interaktionsarten im Kontext *eines* Interfacetyps zusammengefaßt werden können, ist für einen derart kombinierten Interfacetyp in einer CORBA-Umgebung - technologiebedingt - eine Aufspaltung in mehrere CORBA-IDL-Interfacedefinitionen notwendig.

Die Praktikabilität dieses in [CoRE I], Kapitel 3 eingeführten Prinzips der Zusammenfassung aller Interaktionsarten wird durch die hier vorgestellten Ableitungsregeln bestätigt. Während für operationale Interaktionselemente eines Interfacetyps nur eine CORBA-IDL-Interfacedefinition produziert wird, die alle im Entwurfsmodell für diesen Interfacetyp definierten Operationen und Attribute enthält, müssen für Signal- und *Continuous-Media*-Interaktionselemente die Relationen *Requires* (Benutzen eines Interfaces) und *Supports* (Bereitstellen eines Interfaces) von COs bezüglich des Interfacetyps besondere Beachtung finden. So werden *unterschiedliche* CORBA-IDL-Definitionen für das Produzieren bzw. Konsumieren von Signalen eines Signaltyps *bezüglich ein und desselben* Interfacetyps im Entwurfsmodell in Abhängigkeit von seiner Verwendung (*Supports* bzw. *Requires*) erzeugt.

Auch der CORBA-*Event*-Service bzw. der CORBA-*Notification*-Service als Basis der Kommunikationskanäle von *CoRE*$_{WARE}$ für Signale sehen die Benutzung unterschiedlicher CORBA-Interfacedefinitionen für das Produzieren und Konsumieren von Signalen vor. Gleiches gilt für die *Source*- und *Sink*-Definitionen, für deren Realisierung die *Continuous-Media-Delivery*-Plattform der Humboldt-Universität zu Berlin herangezogen wurde. Auch hier sind plattformspezifische Interfacedefinitionen für das Versenden (*Source*) bzw. Empfangen (*Sink*) von *Continuous-Media*-Interaktionselementen definiert.

2.1.5.1 Interfacetyp und Abbildung operationaler Interaktionselemente

Für *Requires*- und *Supports*-Relationen im Kontext eines Interfacetyps, der nur Operationen und Attribute enthält, wird also nur ein und dieselbe CORBA-IDL-Interfacedefinition in beiden Fällen genutzt, während für Signal- und *Continuous-Media*-Interaktionselemente unterschiedliche CORBA-IDL-Interfacedefinitionen für *Requires*- bzw. *Supports*-Relationen produziert und angewendet werden.

(Regel 9) Für eine Instanz des Konzeptes Interfacetyp (Interfacetyp im Entwurfsmodell) wird eine CORBA-IDL-Interfacedefinition gleichen Namens produziert, die alle definierten Operationen, Attribute und lokale Definitionen (Datentypen, Ausnahmen) enthält. Die Abbildung von Operationen und Attributen erfolgt entsprechend der CORBA-IDL-Semantik.

Für den operationalen Anteil der Interaktionselemente eines Interfacetyps im Entwurfsmodell wurde also eine kanonische Repräsentation in CORBA-IDL gefunden. Die Ableitungsregel für die operationale Interaktion folgt dem UML-Profil für CORBA [OMG CORBA P].

(Beispiel 6) Sei ***I*** ein Interfacetyp im Entwurfsmodell im Namensraum ***M1***. Sei weiterhin eine Operation ***op*** ohne Rückgabetyp und Parameter im Kontext dieses Interfacetyps definiert, die die in *Beispiel 3* definierte Ausnahme ***E*** auslösen kann. ***I*** besitze weiterhin ein *Readonly*-Attribut ***attr*** vom Typ ***S*** aus *Beispiel 2*. Dann werden folgende CORBA-IDL-Definitionen produziert:

```
module M1 {
  interface I {
    readonly attribute ::M1::S attr;
    void op () raises ( ::M1::E );
  };
};
```

2.1.5.2 Interfacetyp mit Signal- oder Continuous-Media-Interaktionselementen

Falls für einen Interfacetyp im Entwurfsmodell eine oder mehrere Instanzen der Konzepte *Produce, Consume, Source* und *Sink* definiert sind, so werden, neben der CORBA-IDL-Interfacedefinition für den operationalen Anteil, weitere CORBA-IDL-Interfacedefinitionen erzeugt. Um Namenskonflikte zu vermeiden, werden diese CORBA-IDL-Interfacedefinitionen (*implizite* Interfacedefinitionen) in zusätzlichen CORBA-IDL-Modulen definiert. Zusätzliche CORBA-IDL-Module bedeutet hier, daß diese CORBA-IDL-Module keine Entsprechung im Sinne eines Namensraumes im Entwurfsmodell haben. Die Namen der CORBA-IDL-Module ergeben sich aus dem Namen des Interfacetyps im Entwurfsmodell sowie den möglichen Rollen *User* (korrespondierend zur *Requires*-Relation) und *Supplier* (korrespondierend zur *Supports*-Relation), die ein CO-Typ bezüglich eines Interfacetyp einnehmen kann.

(Regel 10) Für jede Instanz des Konzeptes Interfacetyp (Interfacetyp im Entwurfsmodell), die ein oder mehrere Interaktionselemente der Typen *Source, Sink, Produce* oder *Consume* definiert, werden zwei CORBA-IDL-Moduldefinitionen erzeugt. Diese sind in der CORBA-IDL-Moduldefinition definiert, in dem die CORBA-IDL-Interfacedefinition vorgenommen wurde, die dem Interfacetyp im Entwurfsmodell entspricht. Die Namen dieser Module ergeben sich aus dem Namen des Interfacetyps im Entwurfsmodell und dem Suffix **__Supply___** korrespondierend zur *Supports*-Relation und **__Use__** korrespondierend zur *Requires*-Relation.

Diese Regel impliziert, daß in einem Entwurfsmodell nicht zwei Modellelemente im gleichen Namensraum definiert sein dürfen, von denen ein Modellelement den Namen des anderen erweitert um die Suffixe ***__Supply__*** bzw. ***__Use__*** trägt. Sofern diese Namenskonvention beachtet wird, können keine Namenskonflikte innerhalb der produzierten CORBA-IDL-Definitionen auftreten, da alle CORBA-IDL-Definitionen für den Anteil der Signal- bzw. *Continuous-Media*-Interaktionsarten in den jeweiligen CORBA-IDL-Modulen mit den Suffixen **__Supply__** bzw. **__Use__** definiert sind.

(Beispiel 7) Falls ein Interfacetyp ***I*** im Entwurfsmodell im Namensraum ***M1*** eine *Produce*-Definition für ***Sig1*** (vgl. *Beispiel 4*) beinhaltet, so werden die folgenden CORBA-IDL-Moduldefinitionen produziert:

```
module M1 {
  interface I {
    // ...
  };
  module I__Supply__ {
    // ...
  };
  module I__Use__ {
    // ...
  };
};
```

Es können nun Regeln zur Abbildung der Interaktionselemente formuliert werden, die über die operationale Interaktion hinausgehen. Dabei ist - wie bereits ausgeführt - die Art der Relation zwischen einem Interfacetyp im Entwurfsmodell, für das ein Interaktionselement spezifiziert ist, und CO-Typen entscheidend. Zunächst werden die Ableitungsregeln für Signalinteraktionselemente eingeführt.

2.1.5.3 Abbildung von Signalinteraktionselementen

Interfacetypen können im Entwurfsmodell *Produce*- bzw. *Consume*-Definitionen für Signaltypen beinhalten. Im Falle einer *Produce*-Definition wird also durch den Anbieter des zugehörigen Interfacetyps potentiell ein Signal des entsprechenden Signaltyps versendet, das durch den Nutzer dieses Interfacetyps empfangen werden kann. Anschaulich beziehen sich die Definitionen *Produce* bzw. *Consume* immer auf den Anbieter des Interfaces, der Nutzer eines Interfaces invertiert diese Rollen.

Das Versenden eines Signals ist technologisch zunächst eine Interaktion zwischen der programmiersprachlichen Realisierung des Artefaktes, das für das Versenden verantwortlich ist, und dem Teil von $CoRE_{WARE}$, der im gleichen örtlichen Ausführungskontext wie die Artefaktrealisierung abgearbeitet wird (z.B. im gleichen Betriebssystemprozeß). Dieser Teil von $CoRE_{WARE}$ stellt Interfaces bereit, die es der Artefaktrealisierung erlauben, Signale zu versenden. Die Definitionen dieser Interfaces werden durch $CoRE_{MAP}$ entsprechend den nachfolgenden Regeln erzeugt.

(Regel 11) Für eine Instanz des Konzeptes Interfacetyp (Interfacetyp im Entwurfsmodell) wird für jede im Entwurfsmodell definierte *Produce*-Definition für einen Signaltyp eine CORBA-IDL-Interfacedefinition mit dem Namen **<Name der produce-Definition>** erzeugt (*Local Interface*), die eine Operation mit dem Namen **push_<Name der produce-Definition>** beinhaltet. Diese Operation hat keinen Rückgabetyp und einen **in**-Parameter vom Typ der CORBA-IDL-Repräsentation des Signaltyps. Diese CORBA-IDL-Interfacedefinition erfolgt im Modul **<Interfacename>__Supply__**.

Da sowohl die Artefaktrealisierung als auch die Realisierung der entsprechend *Regel 11* produzierten Interfacedefinitionen im gleichen örtlichen Ausführungskontext abgearbeitet werden, wurde das Konzept von lokalen CORBA-IDL-Interfacedefinitionen (**local interface**) - definiert in [OMGCCM I] - eingesetzt. Dieses Prinzip gestattet eine *normierte* programmiersprachliche Abbildung von nur im Ausführungskontext sichtbaren Interfaces.

Die entsprechend *Regel 11* erzeugten CORBA-IDL-Interfacedefinitionen werden durch die Artefaktimplementierung, d.h. durch programmiersprachliche Konstrukte, die durch das Konzept Artefakt abstrahiert werden, genutzt. Es ist folglich eine Lösung zu erarbeiten, mit dessen Hilfe Artefaktimplementierungen konkrete Mechanismen von $CoRE_{WARE}$ zur Ausführungszeit ermitteln können, die diese generierten Interfacedefinitionen realisieren. Zunächst wird durch *Regel 11* für jede *Produce*-Definition eine CORBA-IDL-Interfacedefinition erzeugt, deren Realisierung der Artefaktimplementierung zur Ausführungszeit bekannt sein muß. Zur Vereinfachung dieser Auflösung wird eine weitere CORBA-IDL-Interfacedefinition als einheitlicher Zugangspunkt zu allen CORBA-Objekten erzeugt, die die nach *Regel 11* produzierten CORBA-IDL-Interfacedefinitionen realisieren.

(Regel 12) Für jede Instanz des Konzeptes Interfacetyp (Interfacetyp in einem Entwurfsmodell), für die *Produce*-Definitionen zu Signaltypen definiert sind, wird eine CORBA-IDL-Interfacedefinition (*Local Interface*) mit dem Namen **<Interfacename>__Supplier** im CORBA-IDL-Modul ***<Interfacename>__Supply__*** erzeugt. Diese produzierte CORBA-IDL-Interfacedefinition enthält **readonly**-Attributdefinitionen für alle entsprechend *Regel 11* im Kontext des Interfacetyps im Entwurfsmodell erzeugten CORBA-IDL-Interfacedefinitionen. Die Attributnamen ergeben sich aus dem Namen der entsprechenden *Produce*-Definition sowie dem Suffix **_supplier**. Zur Ausführungszeit stellt $CoRE_{WARE}$ CORBA-Objekte, die diese lokale Interfacedefinition unterstützen, zur Verfügung.

Die Implementierung eines CO-Typs nutzt zum Versenden von Signalen CORBA-Objekte, die die entsprechend *Regel 11* erzeugten **push**-Operationen implementieren. Den Zugang zu diesen CORBA-Objekten erhält sie unter Nutzung eines weiteren CORBA-Objektes, das diese Objekte via Attributdefinition zur Verfügung stellt (vgl. *Regel 12*).

(Beispiel 8) Sei ***I*** ein Interfacetyp im Entwurfsmodell im Namensraum ***M1***. Sei weiterhin ***Sig1*** der in *Beispiel 4* definierte Signaltyp. Im Kontext von ***I*** seien zwei *Produce*-Definitionen mit den Namen

send_1_Sig und ***send_2_Sig*** basierend auf ***Sig1*** definiert. Für die *Supplier*-Rolle (d.h. zur Nutzung durch CO-Typimplementierungen mit einer *Supports*-Relation zu ***I***) werden folgende CORBA-IDL-Definitionen produziert:

```
module M1 {
  module I__Supply__ {
      local interface send_2_Sig  {
      void push_send_2_Sig ( in ::M1::Sig1 signal );
    };

    local interface send_1_Sig {
      void push_send_1_Sig ( in ::M1::Sig1 signal );
    };

    local interface I__Supplier : ::ComponentModel::SupplierBase {
      readonly attribute send_1_Sig send_1_Sig_supplier;
      readonly attribute send_2_Sig send_2_Sig_supplier;
    };
  };
};
```

Die in *Beispiel 8* präsentierte Situation ist in Abb. 5 illustriert. Die Implementierungen der lokalen CORBA-IDL-Interfacedefinitionen **send_2_Sig** und **send_1_Sig** werden durch $CoRE_{MAP}$ geliefert, die auf Signalkommunikationsmechanismen von $CoRE_{WARE}$ aufsetzen. Neben der Bereitstellung dieser Objekte wird der Zugangspunkt zu diesen ebenfalls durch $CoRE_{MAP}$ über produzierten Implementierungscode für **I__Supplier** erzeugt.

Mit der Definition der obigen Regeln ist festgelegt, auf welche Weise sich *Produce*-Definitionen für Signaltypen im Kontext von Interfacetypen im Entwurfsmodell auf die externe Sicht auf einen CO-Typ auswirken, für den eine *Supports*-Relation zu diesen Interfacetypen definiert wurde. Im folgenden werden Ableitungsregeln von $CoRE_{MAP}$ definiert, die auf CO-Typen angewendet werden, für die eine *Requires*-Relation zu diesen Interfacetypen besteht.

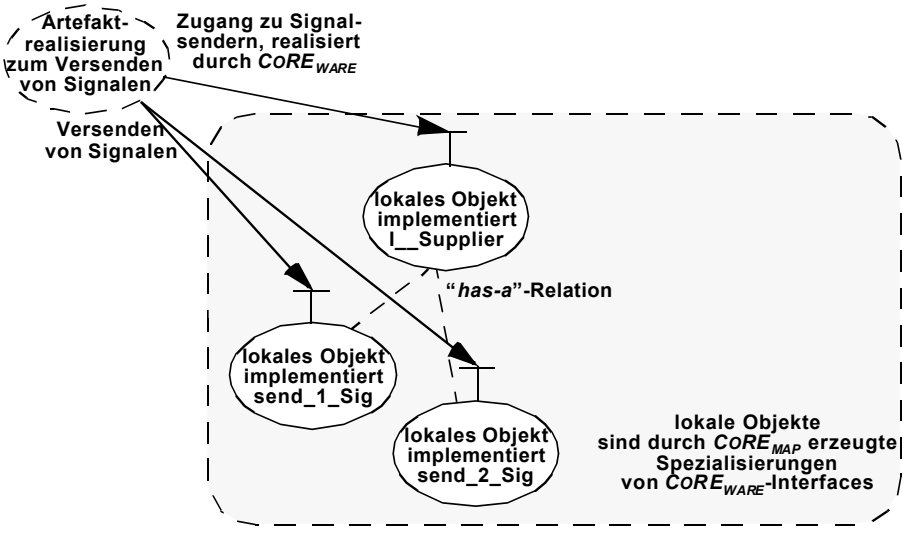


Abb. 5 CO-Typrealisierung und Implementierungen lokaler Objekte für das Versenden von Signalen

(Regel 13) Für jede Instanz des Konzeptes *Produce*-Definition (*Produce*-Definition im Entwurfsmodell) zwischen einem Signaltyp und einem Interfacetyp wird im CORBA-IDL-Modul **<Interfacename>__Use__** eine CORBA-IDL-Interfacedefinition **<Name der produce-Definition>_consumer>** erzeugt. Diese CORBA-IDL-Interfacedefinition enthält eine Operation **push_<Name der produce-Definition>**, die keinen Rückgabetyp und einen **in**-Parameter vom Typ der CORBA-IDL-Repräsentation des produzierten Signaltyps hat (entsprechend *Regel 5*). Die produzierte CORBA-IDL-Interfacedefinition ist als Spezialisierung der CORBA-IDL-Basisinterfacedefinition **ComponentModel::ConsumerBase** definiert.

Zur Ausführungszeit verknüpft $CoRE_{WARE}$ CORBA-Objekte, die dieses CORBA-IDL-Interface unterstützen, mit entsprechenden Signalkommunikationskanälen (z.B. CORBA-*Event*-Kanal, CORBA-*Notification*-Kanal). Die Implementierung eines CO-Typs, für den im Entwurfsmodell eine *Requires*-Relation zu dem den Signaltyp produzierenden Interfacetyp formuliert wurde, empfängt über diese CORBA-Objekte Signale.

In $CoRE_{MAP}$ wird für CORBA-Objekte, die die entsprechend *Regel 13* produzierte CORBA-IDL-Interfacedefinitionen implementieren, nicht das Konzept lokaler CORBA-IDL-Interfacedefinitionen eingesetzt. Der Grund liegt darin, daß $CoRE_{WARE}$ diese CORBA-IDL-Interfacedefinitionen lokationstransparent (d.h. ohne Rücksicht auf räumliche Verteilung) direkt benutzen kann, um Signale typsicher auszuliefern. Direkte Nutzung heißt dabei beispielsweise ohne Verwendung von CORBA-*Event*-Service oder CORBA-*Notification*-Service. Falls die eben beschriebenen, für das Signalkommunikationsmodell standardisierten Services genutzt werden, ist u. U. eine Transformation zwischen dem für den Transport zu wählenden CORBA-Datentyp **any** und dem **valuetype**-Konstrukt, der für den Signaltyp entsprechend *Regel 5* produziert wurde, vorzunehmen. Diese Transformation entfällt bei der direkten Nutzung der entsprechend *Regel 13* produzierten CORBA-IDL-Interfacedefinition.

Die Definition aller entsprechend *Regel 13* produzierten CORBA-IDL-Interfacedefinitionen als Spezialisierung von **ComponentModel::ConsumerBase** dient der transparenten Anbindung dieser CORBA-IDL-Interfaces an entkoppelte Kommunikationskanäle der *Component-Support*-Plattform. Wird beispielsweise der CORBA-*Event*-Service als Grundlage der Kommunikationskanäle eingesetzt, so wird **ComponentModel::ConsumerBase** gerade als Spezialisierung der in [OMG CORBAES] normierten Basisinterfacedefinitionen für *Event*-Konsumenten definiert[1]:

```
interface ConsumerBase
    : CosEventComm::PushConsumer
  {};
```

(Beispiel 9) Für die in *Beispiel 8* dargestellte Situation ergeben sich die folgenden CORBA-IDL-Definitionen im Kontext von CO-Typen, die eine *Requires*-Relation zum Interfacetyp ***I*** definieren:

```
module M1 {
  module I__Use__ {
    interface send_1_Sig_consumer : ::ComponentModel::ConsumerBase
    {
      void push_send_1_Sig ( in Sig1 signal );
    };
    interface send_2_Sig_consumer : ::ComponentModel::ConsumerBase
    {
      void push_send_2_Sig ( in Sig1 signal );
    };
  };
};
```

Bei der Nutzung des CORBA-*Event*-Service bzw. CORBA-*Notification*-Service (d.h. im Falle der nicht-direkten Signalsauslieferung) werden durch $CoRE_{WARE}$ *alle* produzierten Signaltypen eines Interfacetyps im Ent-

1. Die äquivalente Interfacedefinition des CORBA-*Notification*-Service ist **CosNotifyComm::PushConsumer**. Diese Interfacedefinition spezialisiert **CosEventComm::PushConsumer**, so daß die Legitimität der Verwendung des CORBA-*Event*-Service anstelle des CORBA-*Notification*-Service innerhalb dieser Arbeit gegeben ist.

wurfsmodell im Kontext *eines* CORBA-*Event*- bzw. CORBA-*Notification*-Kanals behandelt. Zur Unterstützung dieses Mechanismus wird durch *CORE*$_{MAP}$ eine weitere CORBA-IDL-Interfacedefinition produziert, das die entsprechend *Regel 13* generierten CORBA-IDL-Interfacedefinitionen mittels Vererbung zusammenfaßt.

(Regel 14) Für jede Instanz des Konzeptes Interfacetyp (Interfacetyp im Entwurfsmodell), das *Produce*-Definitionen für Signaltypen enthält, wird im CORBA-IDL-Modul **<Interfacename>__Use__** eine CORBA-IDL-Interfacedefinition mit dem Namen **<Interfacename>__Consumer** erzeugt. Diese CORBA-IDL-Interfacedefinition hat alle diejenigen CORBA-IDL-Interfacedefinitionen als Basis, die entsprechend *Regel 13* für den Interfacetyp im Entwurfsmodell erzeugt wurden.

(Beispiel 10) Entsprechend *Regel 14* werden die in *Beispiel 9* dargestellten CORBA-IDL-Interfacedefinitionen folgendermaßen zusammengefaßt:

```
module M1 {
  module I__Use__ {
    interface I__Consumer
      : send_1_Sig_consumer,
        send_2_Sig_consumer
    {};
  };
};
```

Die Situation aus *Beispiel 10* ist in Abb.6 illustriert. *CORE*$_{MAP}$ stellt Mechanismen zur Implementierung der produzierten Interfacedefinitionen zum Signalempfang bereit, wobei sowohl der ungetypte als auch der getypte Empfang von Signalen unterstützt wird. Ungetypter Signalempfang bedeutet Benutzung des

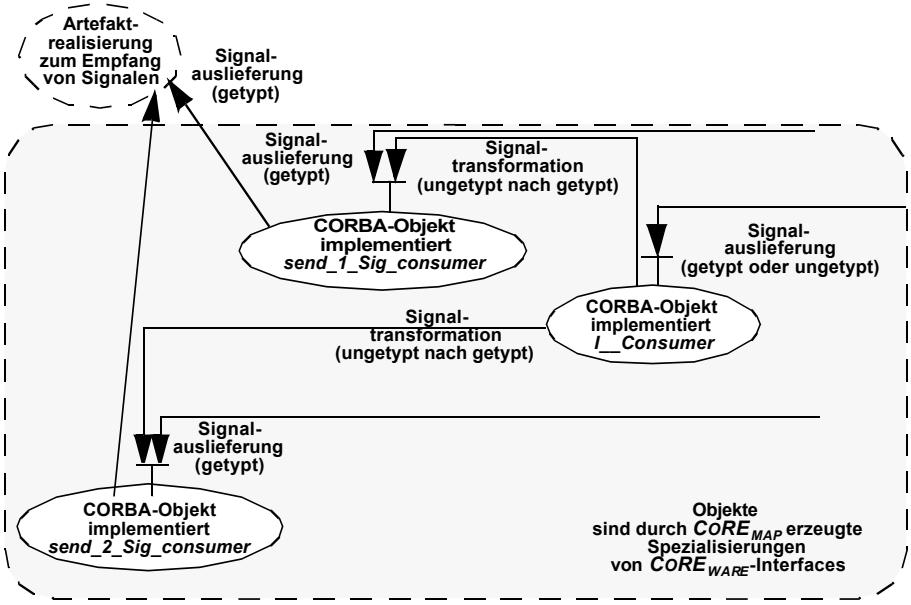


Abb. 6 CO-Typrealisierung und Implementierungen von CORBA-Objekten für den Signalempfang

CORBA-Datentyps **any** bzw. des CORBA-*Notification*-Servicedatentyps **StructuredEvent**. In diesem Fall stellt die durch *CORE*$_{MAP}$ generierte Implementierung für die produzierte CORBA-IDL-Interfacedefinition **I__Consumer** die Transformation in ein getyptes Signal, in diesem Fall in **Sig1** bereit, das dann an die Artefaktrealisierung zum Signalempfang innerhalb der Implementierung des CO-Typs ausgeliefert wird. Werden die

produzierten CORBA-IDL-Definitionen von *Beispiel 9* und *Beispiel 10* zusammengefaßt, so ergeben sich unterschiedliche Szenarien für die eingesetzten Mechanismen zur Signalkommunikation (vgl. Abb.7):

- Die Signalübertragung kann in getypter Form zwischen der Implementierung von **send_1_Sig** und **send_1_Sig_consumer** bzw. zwischen **send_2_Sig** und **send_2_Sig_consumer** erfolgen (Szenario 1 in Abb. 7) oder
- die Implementierungen von **send_1_Sig** und **send_2_Sig** können die **any**- bzw. **StructuredEvent**-Repräsentationen des zu transportierenden Signals des Signaltyps **Sig1** an einen CORBA-*Event*- bzw. CORBA-*Notification*-Kanal ausliefern, der wiederum diese Repräsentationen an das CORBA-Objekt **I__Consumer** ausliefert (Szenario 2 in Abb. 7). Die Implementierung von **I__Consumer** ist dann verantwortlich für die Transformation der **any**- bzw. **StructuredEvent**-Darstellung von **Sig1** in die getypte CORBA-**valuetype**-Repräsentation von **Sig1** sowie die Auslieferung des Signals an die Artefaktrealisierung für den Empfang von **Sig1** innerhalb der CO-Typimplementierung.

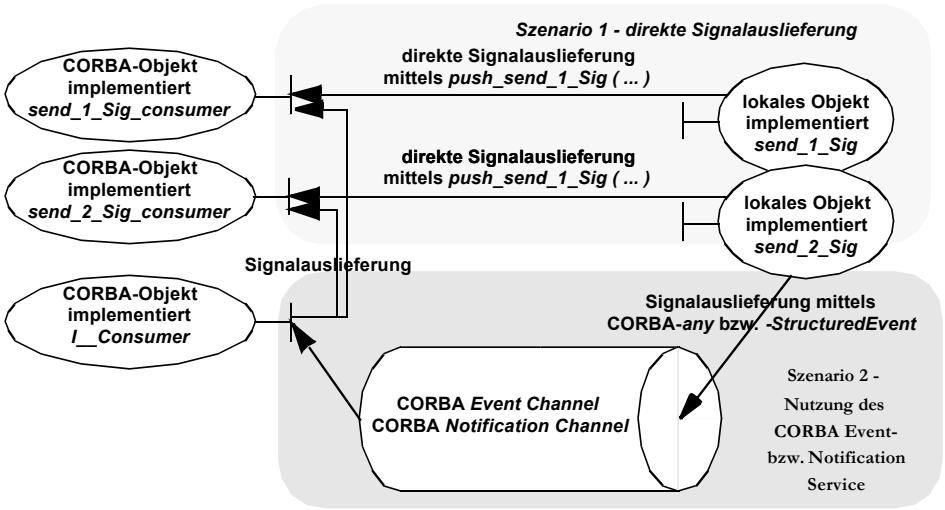


Abb. 7 Szenarien zur Anwendung der Mechanismen zur Signalkommunikation

Die bisher beschriebenen Ableitungsregeln sind anwendbar auf *Produce*-Definitionen im Kontext eines Interfacetyps im Entwurfsmodell. Dabei werden alle für die Anbieter von Interfaces dieses Interfacetyps (Beachtung der *Supports*-Relation) produzierten CORBA-IDL-Definitionen im CORBA-IDL-Modul **<Interfacename>__Supply__** vorgenommen, die für die Nutzer (Beachtung der *Requires*-Relation) relevanten Definitionen im CORBA-IDL-Modul **<Interfacename>__Use__**. Offensichtlich können für *Consume*-Definitionen die gleichen Ableitungsregeln eingesetzt werden, jedoch muß die Zuordnung der produzierten CORBA-IDL-Definitionen zu den CORBA-IDL-Modulen **<Interfacename>__Supply__** bzw. **<Interfacename>__Use__** vertauscht werden.

(Regel 15) Für jede Instanz des Konzeptes Interfacetyp (Interfacetyp im Entwurfsmodell) werden für jede definierte *Consume*-Definition für einen Signaltyp die Ableitungsregeln *Regel 11* - *Regel 14* angewendet, jedoch wird die Zuordnung der produzierten CORBA-IDL-Definitionen zu den CORBA-IDL-Modulen **<Interfacename>__Use__** und **<Interfacename>__Supply__** invertiert.

Die Definition von *Regel 15* in Kombination mit *Regel 11* bis *Regel 14* erzeugt symmetrische CORBA-IDL-Definitionen für *Produce*- und *Consume*-Definitionen in den entsprechenden CORBA-IDL-Modulen **<Interfacename>__Supply__** und **<Interfacename>__Use__**. $CORE_{CEPT}$ erlaubt die *Kombination* von *Produce*- und *Consume*-Definitionen im Kontext *eines* Interfacetyps im Entwurfsmodell. $CORE_{MAP}$ sichert auch dann unter

Anwendung von *Regel 11* bis *Regel 15* die korrekte Abbildung dieser Kombinationen nach CORBA-IDL, wie das folgende Beispiel verdeutlicht.

(Beispiel 11) Sei ***I*** ein Interfacetyp im Entwurfsmodell im Namensraum ***M1***, für den eine *Produce*-Definition unter dem Namen ***send_Sig*** sowie eine *Consume*-Definition unter dem Namen ***receive_Sig*** basierend auf dem in *Beispiel 4* definierten Signaltyp ***Sig1*** erklärt sind. Die Kombination von *Consume*- und

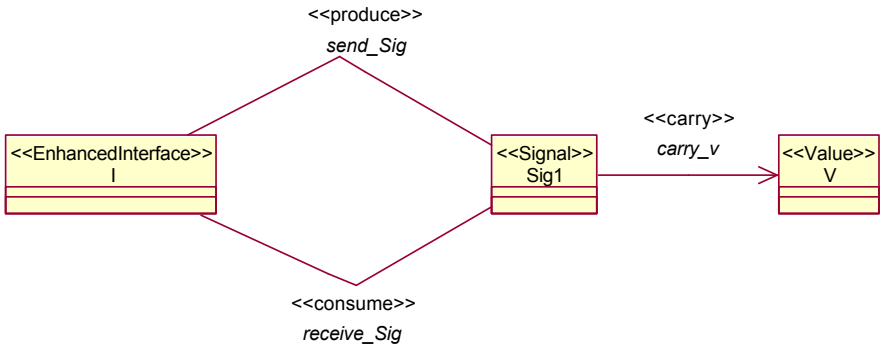


Abb. 8 Beispiel für Signalinteraktion

Produce-Definitionen zwischen dem Interfacetyp und dem Signaltyp verdeutlicht Abb. 8 unter Verwendung der in [CoRE II], Kapitel 5 definierten Notation. Entsprechend *Regel 11* bis *Regel 14* werden für Anbieter dieses Interfacetyps (*Supports*-Relation) die folgenden CORBA-IDL-Definitionen produziert:

```
module M1 {
  interface I {};

  module I__Supply__ {
    interface receive_Sig_consumer : ::ComponentModel::ConsumerBase {
      void push_receive_Sig ( in ::M1::Sig1 signal );
    };

    interface I__Consumer : receive_Sig_consumer {};

    local interface send_Sig {
      void push_send_Sig (
        in ::M1::Sig1 signal);
    };

    local interface I__Supplier : ::ComponentModel::SupplierBase {
      readonly attribute ::M1::I__Supply__::send_Sig send_Sig_supplier;
    };
  };
};
```

Für einen Nutzer dieses Interfacetyps (*Requires*-Relation) werden die folgenden CORBA-IDL-Definitionen im CORBA-IDL-Module **<Interfacename>__Use__** erzeugt:

```
moule M1 {
  module I__Use__ {
    interface send_Sig_consumer : ::ComponentModel::ConsumerBase {
      void push_send_Sig (
        in ::M1::Sig1 signal);
    };

    interface I__Consumer
      : send_Sig_consumer {};
```

```
    local interface receive_Sig {
      void push_receive_Sig (
        in ::M1::Sig1 signal);
    };

    local interface I__Supplier {
      readonly attribute ::M1::I__Use__::receive_Sig receive_Sig_supplier;
    };
  };
};
```

2.1.5.4 Abbildung von Continuous-Media-Interaktionselementen

Neben der operationalen und Signalinteraktion unterstützt *CORE* die Definition von *Continuous-Media*-Interaktionselementen für Interfacetypen in Entwurfsmodellen. Während für Signalinteraktionen normierte CORBA-Services, wie der CORBA-*Event*-Service oder der CORBA-*Notification*-Service eingesetzt werden können, werden für *Continuous-Media*-Interaktionen die in Abschnitt 1.2.2 beschriebenen Konzepte verwendet. Der Grund für die Nutzung einer proprietären, d.h. nicht normierten, Technologie besteht darin, daß die Verwendung von [OMG AVStreams] als normierte Lösung für die Steuerung von Audio- und Videoströmen nicht für das allgemeinere Konzept von Medien als Meta-Typ einsetzbar ist - die normierte Lösung ist nicht allgemein anerkannt, es sind keine unterstützenden Produkte verfügbar.

Implementierungen von CO-Typen, die *Continuous-Media*-Interaktionselemente an ihren Interfaces anbieten bzw. nutzen, stellen CORBA-Objekte zur Ausführungszeit bereit, die die in Abschnitt 1.2.2 dargelegte *Media-Manager*- (*Supplier*-Rolle, Definition *source* zwischen Interfacetyp und Medienmengen) bzw. *Media-Device*-Funktionalität (*User*-Rolle, Definition *source* zwischen Interfacetyp und Medienmenge) für die verwendeten Medien implementieren. Somit werden für *Source*- bzw. *Sink*-Definitionen zwischen Interfacetypen und Medienmengen Spezialisierungen der in $CORE_{WARE}$ spezifizierten CORBA-IDL-Interfacedefinition **::ComponentModel::MediaDevice** bzw. **::ComponentModel::MediaManager** produziert. Für allgemeine Anwendungsfälle bietet die *Multimedia-Content-Delivery*-Plattform vorgefertigte Implementierungen an, die in Artefaktimplementierungen direkt verwendet werden können.

Analog zur Darstellung der Ableitungsregeln für Signalinteraktion werden zunächst *Source*-Definitionen, nachfolgend *Sink*-Definitionen und die Kombination aus *Source*- und *Sink*-Definitionen untersucht.

(Regel 16) Für jede Instanz des Konzeptes *Source*-Definition (*Source*-Definition im Entwurfsmodell) basierend auf einer Medienmenge im Kontext eines Interfacetyp wird eine CORBA-IDL-Interfacedefinition **<Name der source-Definition>__Source** im CORBA-IDL-Modul **Interfacename>__Supply__** produziert. Diese CORBA-IDL-Interfacedefinition enthält für jedes aggregierte Medium eine Operation. Der Name dieser Operation entspricht dem Namen der Aggregation, der Rückgabetyp ist **::ComponentModel::MediaManager**.

CORBA-Objekte, die diese CORBA-IDL-Interfacedefinition im Rahmen einer CO-Typimplementierung anbieten, werden durch $CORE_{WARE}$ genutzt, um den Zugang zu einem geeigneten *Media*-Manager zu erhalten, der den Zugriff auf das entsprechende Medium steuert.

(Regel 17) Für jede Instanz des Konzeptes Interfacetyp (Interfacetyp im Entwurfsmodell), die mindestens eine *Source*-Definition zu einer Medienmenge enthält, wird eine CORBA-IDL-Interfacedefinition mit dem Namen **<Interfacename>__Sources** im CORBA-IDL-Modul **<Interfacename>__Supply__** generiert. Diese CORBA-IDL-Interfacedefinition besitzt für jede entsprechend *Regel 16* produzierte CORBA-IDL-Interfacedefinition ein **readonly**-Attribut.

CORBA-IDL-Interfacedefinitionen, die entsprechend *Regel 17* produziert wurden, fassen die für die *Source*-Definitionen im Kontext eines Interfacetyps im Entwurfsmodell generierten CORBA-IDL-Definitionen zusammen. Damit wird $CORE_{WARE}$ der einheitliche Zugang zu allen **MediaManager**-Objekten gestattet.

(Beispiel 12) Sei ***I*** ein Interfacetyp im Entwurfsmodell im Namensraum ***M1***. Sei ***M*** eine Medienmenge im Entwurfsmodell, die das Medium ***Audio*** unter dem Namen ***audio*** aggregiert. Für ***I*** und ***M*** sei eine *Source*-Definition unter dem Namen ***se*** definiert. Diese Situation verdeutlicht Abb. 9. Dann werden für

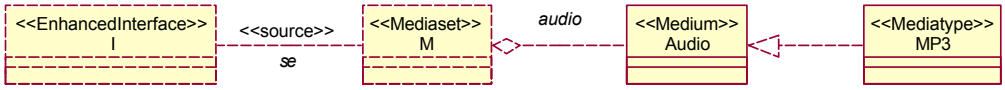


Abb. 9 Beispielszenario für Ableitungsregeln für source-Definitionen

Anbieter dieses Interfaces die folgenden CORBA-IDL-Definitionen erzeugt:

```
module M1
{
  module I__Supply__
  {
    interface se__Source
    {
      ::ComponentModel::MediaManager se_audio ( );
    };
    interface I__Sources
    {
      readonly attribute ::M1::I__Supply__::se__Source se;
    };
  };
};
```

In der Gegenrichtung, d.h. zum Empfang von kontinuierlichen Medien wird die zu **::ComponentModel::MediaManager** komplementäre CORBA-IDL-Interfacedefinition **::ComponentModel::MediaDevice** eingesetzt. Eine ein kontinuierliches Medium empfangende Implementierung eines CO-Typs muß $CORE_{WARE}$ entsprechend Abschnitt 1.2.2 ein CORBA-Objekt zur Verfügung stellen, welches seinerseits das durch die Plattform geforderte CORBA-IDL-Interface **MediaDevice** anbietet. Demzufolge werden für den Nutzer (*Requires*-Relation) eines Interfacetyps im Entwurfsmodell, für den *Source*-Definitionen definiert sind, entsprechende CORBA-IDL-Interfacedefinitionen zum Empfang von Medien unter Verwendung der folgenden Regel produziert.

(Regel 18) Für jede Instanz von *Source*-Definition im Kontext eines Interfacetyps im Entwurfsmodell wird im Modul **<Interfacename>__Use__** eine CORBA-IDL-Interfacedefinition mit dem Namen **<Name der source-Definition>__Sink** erzeugt. Diese stellt für jedes der aggregierten Medien eine Operation mit dem Namen der Aggregation bereit. Der Rückgabetyp dieser Operation ist **::ComponentModel::MediaDevice**.

Analog zur Definition der zusammenfassenden CORBA-IDL-Interfacedefinition **<Interfacename>__Sources** für Anbieter eines Interfacetyps im Entwurfsmodell wird auch für alle Senken eine entsprechende Zusammenfassung erzeugt.

(Regel 19) Für jede Instanz des Konzeptes Interfacetyp im Entwurfsmodell, das mindestens eine *Source*-Definition zu einer Medienmenge definiert, wird eine CORBA-IDL-Interfacedefinition mit dem Namen **<Interfacename>__Sinks** im CORBA-IDL-Modul **<Interfacename>__Use__** generiert. Diese CORBA-IDL-Interfacedefinition besitzt für jede entsprechend *Regel 18* produzierte CORBA-IDL-Interfacedefinition ein **readonly**-Attribut.

In Erweiterung von *Beispiel 12* können nun im folgenden Beispiel die produzierten CORBA-IDL-Definitionen für die Nutzung eines Interfacetyps dargestellt werden.

(Beispiel 13) Für Nutzer des in *Beispiel 12* definierten Interfacetyps ***I*** werden im CORBA-IDL-Modul **<Interfacename>__Use__** die folgenden CORBA-IDL-Definitionen erzeugt:

```
module M1 {
  module I__Use__ {
    interface se__Sink
    {
      ::ComponentModel::MediaDevice se_audio ( );
    };
    interface I__Sinks
    {
      readonly attribute ::M1::I__Use__::se__Sink se;
    };
  };
};
```

Analog zu den Ableitungsregeln für Signalinteraktionen resultieren die für *Sink*-Definitionen zwischen Medienmengen und Interfacetypen zu produzierenden CORBA-IDL-Definitionen aus der umgekehrten Zuordnung der durch *Regel 16* bis *Regel 19* produzierten Definitionen zu den CORBA-IDL-Modulen **<Interfacename>__Supply__** bzw. **<Interfacename>__Use__**. Dies wird durch die folgende Regel präzisiert.

(Regel 20) Für eine Instanz des Konzeptes Interfacetyp im Entwurfsmodell werden für jede definierte *Sink*-Definition zu einer Medienmenge die gleichen Generierungsregeln wie für *Source*-Definitionen angewendet (*Regel 16* bis *Regel 19*), jedoch wird die Zuordnung der produzierten CORBA-IDL-Interfacedefinitionen zu den CORBA-IDL-Modulen **<Interfacename>__Supply__** und **<Interfacename>__Use__** invertiert.

Ebenso wie im Falle von Signalinteraktion sind auch bei der Definition von *Continuous-Media*-Interaktionselementen Kombinationen aus *Source*- und *Sink*-Definitionen im Kontext eines Interfacetyps im Entwurfsmodell interessant. Eine solche Situation verdeutlicht das folgende Beispiel, illustriert in Abb.10.

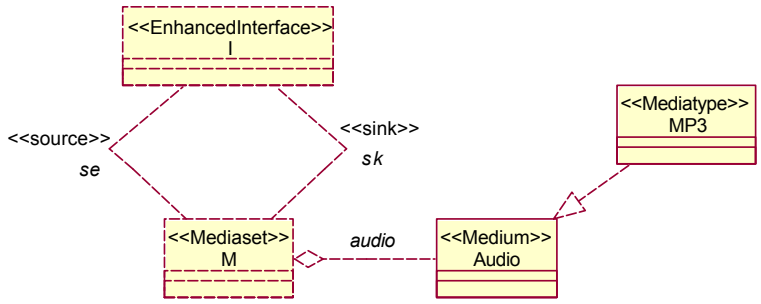


Abb. 10 Beispiel für Continuous-Media-Interaktion

(Beispiel 14) Die in *Beispiel 13* dargestellte Situation sei insofern erweitert, als das der Interfacetyp ***I*** neben der *Source*-Definition eine *Sink*-Definition ***sk*** zu der Medienmenge ***M*** definiert. Dann werden entsprechend *Regel 16* bis *Regel 20* die folgenden CORBA-IDL-Definitionen erzeugt:

```
module M1
{
  module I__Supply__ {
    interface sk__Sink {
      ::ComponentModel::MediaDevice sk_audio ( );
    };

    interface I__Sinks {
      readonly attribute ::M1::I__Supply__::sk__Sink sk;
    };
```

```
      interface se__Source {
        ::ComponentModel::MediaManager se_audio ( );
      };

      interface I__Sources {
        readonly attribute ::M1::I__Supply__::se__Source se;
      };
    };
    module I__Use__ {
      interface sk__Source {
        ::ComponentModel::MediaManager sk_audio ( );
      };

      interface I__Sources {
        readonly attribute ::M1::I__Use__::sk__Source sk;
      };

      interface se__Sink {
        ::ComponentModel::MediaDevice se_audio ( );
      };

      interface I__Sinks {
        readonly attribute ::M1::I__Use__::se__Sink se;
      };
    };
  };
```

2.1.5.5 *Vererbung von Interfacetypen*

Interfacetypen in Entwurfsmodellen können - wie in [CoRE I], Kapitel 3 ausgeführt - in einer Vererbungshierarchie stehen. Diese Vererbungshierarchie wird in eine entsprechende Vererbungshierarchie der für die Interfacetypen produzierten CORBA-IDL-Interfacedefinitionen überführt.

(Regel 21) CORBA-IDL-Interfacedefinitionen, die für eine Instanz des Konzeptes Interfacetyp produziert werden, der von einem oder mehreren Instanzen von Interfacetyp ableitet, werden als Spezialisierung derjenigen CORBA-IDL-Interfacedefinitionen definiert, die für die Basis-Interfacetypen des Interfacetyps im Entwurfsmodell erzeugt wurden.

2.1.6 *CO-Typ*

Definitionen von CO-Typen in einem *CoRE*-basierten Entwurfsmodell erfassen die Erklärung des CO-Typs selbst sowie gegebenenfalls von ihm definierte Operationen und Attribute. Relationen des Typs *Supports* und *Requires*, die zwischen CO-Typen und Interfacetypen definiert sein können, haben erst im Zusammenhang mit der Konfigurationssicht Auswirkungen auf die Ableitungsregeln von $CoRE_{MAP}$ und werden im Kontext der Konzepte der Konfigurationssicht beschrieben. $CoRE_{CEPT}$ definiert CO-Typen als Spezialisierung des Konzeptes Interfacetyp - folgerichtig werden für CO-Typen CORBA-IDL-Interfacedefinitionen erzeugt. $CoRE_{WARE}$ spezifiziert eine CORBA-IDL-Interfacedefinition für CO-Typen, deren Realisierung Basisfunktionalität zur Interfacenavigation, Identifizierung und Reflexion bereitstellt.

(Regel 22) Instanzen des Konzeptes CO-Typ im Entwurfsmodell werden auf CORBA-IDL-Interfacedefinitionen gleichen Namens abgebildet. Diese CORBA-IDL-Interfacedefinitionen enthalten CORBA-IDL-Definitionen für alle Operationen und Attribute, die der CO-Typ definiert. Für CO-Typen, die nicht Spezialisierungen eines anderen CO-Typs sind, leiten die produzierten CORBA-IDL-Interfacedefinitionen von **::ComponentModel::ComponentBase** ab.

Die Definition von **::ComponentModel::ComponentBase** beinhaltet Operationsdefinitionen zur Interfacenavigation, Identifizierung und Reflexion:

```
module ComponentModel {
  valuetype ComponentKey
  {
    public string the_key;
    boolean equal ( in ComponentKey key );
    factory create ();
  };
  interface ComponentBase
  {
    readonly attribute ComponentKey ;
    // operations for configuration and management
    //
  };
};
```

Zunächst wird hier derjenige Teil der allgemeinen **ComponentBase**-Interfacedefinition eingeführt, der die CO-Identität und -Identifizierung realisiert. Dazu wird die CORBA-IDL-**valuetype**-Definition **ComponentKey** definiert, die eine eindeutige Identifizierung über das *Public-Member*-Element **the_key** erlaubt. Diese Identifizierung kann beispielsweise durch die Nutzung von *Universal Unique IDs* (*UUIDs,* [MS COM]) realisiert werden, wie durch die Implementierung von *CORE*$_{WARE}$ gezeigt. Mit Hilfe der in der **ComponentKey**-Definition enthaltenen Operation **equal** kann überprüft werden, ob zwei **ComponentKey**-Instanzen zu ein und demselben CO gehören.

(Beispiel 15) Sei ***CO*** ein CO-Typ im Entwurfsmodell im Namensraum ***M1***. ***CO*** definiere eine Operation ***op*** ohne Parameter und Rückgabetyp, sowie ein Attribut ***attr*** vom Typ ***string***. Dann werden die folgenden CORBA-IDL-Definitionen produziert:

```
module M1
{
  interface CO : ::ComponentModel::ComponentBase {
    attribute string attr;
    void op ( );
  };
};
```

Für eine Instanz des Konzeptes CO-Typ können - wie in [CoRE I], Abschnitt 3.1.6 ausgeführt - Generalisierungsbeziehungen zu anderen Instanzen von CO-Typ existieren. Diese werden auf Vererbung der die CO-Typen repräsentierenden CORBA-IDL-Interfacedefinition abgebildet.

(Regel 23) Die CORBA-IDL-Interfacedefinition, die für eine Instanz des Konzeptes CO-Typ im Entwurfsmodell produziert wird, der von einer oder mehreren Instanzen von CO-Typ ableitet, wird als Spezialisierung derjenigen CORBA-IDL-Interfacedefinitionen definiert, die für die Basis-CO-Typen des CO-Typs im Entwurfsmodell erzeugt wurden.

(Beispiel 16) Sei ***CODerived*** ein CO-Typ im Entwurfsmodell im Namensraum ***M1***, der den in *Beispiel 15* definierten CO-Typ ***CO*** spezialisiert. Dann werden für ***CODerived*** die folgenden CORBA-IDL-Definitionen erzeugt:

```
module M1
{
  interface CODerived : CO
    {};
};
```

2.1.7 CO-Fabriken

Der Darstellung von $CORE_{CEPT}$ folgend werden diejenigen Mechanismen, die zur Erzeugung von COs aus den Definitionen von CO-Typen dienen, nicht in einem *CORE*-basierten Entwurfsmodell erfaßt. Diese werden durch $CORE_{MAP}$ implizit erzeugt und auf konkrete Mechanismen von $CORE_{WARE}$ abgebildet.

$CORE_{WARE}$ unterstützt das Erzeugen bzw. Auffinden eines COs durch die Umgebung dieses CO (vgl. Abschnitt 1.3). Damit diese Mechanismen sowohl generisch als auch abhängig von einem konkreten CO-Typ realisiert werden können, muß die externe Sicht auf einen CO-Typ durch CORBA-IDL-Definitionen für entsprechende CO-Fabriken angereichert werden. Dazu wird die folgende Ableitungsregel definiert:

(Regel 24) Für eine Instanz des Konzeptes CO-Typ (CO-Typdefinition im Entwurfsmodell) wird eine CORBA-IDL-Interfacedefinition mit dem Namen **<CO-Typname>COFactory** im gleichen CORBA-Modul erzeugt, in dem sich auch die die CO-Typdefinition selbst repräsentierende CORBA-IDL-Interfacedefinition befindet. Die produzierte CORBA-IDL-Interfacedefinition bietet eine Operation mit dem Namen **create_<CO-Typname>** an. Diese Operation besitzt diejenige CORBA-IDL-Interfacedefinition als Rückgabetyp, die die CO-Typdefinition repräsentiert.

Diese Operation wird von der Umgebung eines COs zur Erzeugung desselben genutzt. Der Rückgabetyp der Instanziierungsoperation **create** ist die CORBA-IDL-Interfacedefinition, die für die korrespondierende CO-Typdefinition im Entwurfsmodell erzeugt wurde. Ein generisches Konfigurations- und Managementwerkzeug kann nicht direkt COs dieses Typs instanziieren, da die konkrete CORBA-IDL-Definition für diesen CO-Typ i.allg. nicht verfügbar ist. Zur Unterstützung eines generischen Mechanismus zur CO-Erzeugung definiert *CORE* eine allgemeine **COFactory**-Interfacedefinition **ComponentModel::CoFactoryBase**:

```
module ComponentModel {
  typedef sequence<ComponentKey> ComponentKeySeq;

  interface CoFactoryBase {
    string get_co_type ();

    ComponentBase generic_create ();

    ComponentBase resolve_co ( in ComponentKey key );
    ComponentKeySeq list_cos ();
  };
};
```

Die Operation **get_co_type** dient der Ermittlung des CO-Typs, der durch die CO-Fabrik unterstützt wird. Der Rückgabewert entspricht dabei dem vollständigen Namen des CO-Typs im Entwurfsmodell. Die Operation **generic_create** gestattet die Instanziierung eines COs ohne Kenntnis des spezifischen CO-Typs. Mit Hilfe dieser Operation ist es einem generischen Konfigurationswerkzeug möglich, COs zu erzeugen. Die Operationen **resolve_co** und **list_cos** dienen dem Auffinden von COs, die bereits durch eine CO-Fabrik instanziiert wurden. CORBA-IDL-Interfacedefinitionen für CO-Fabriken spezialisieren die CORBA-IDL-Interfacedefinition **CoFactoryBase** entsprechend der folgenden Regel:

(Regel 25) CORBA-IDL-Interfacedefinitionen für CO-Typfabriken, deren korrespondierender CO-Typ im Entwurfsmodell keine Basis-CO-Typen besitzt, spezialisieren die CORBA-IDL-Interfacedefinition **::ComponentModel::CoFactoryBase**, anderenfalls werden die CORBA-IDL-Interfacedefinitionen spezialisiert, die für die Basis-CO-Typen erzeugt wurden.

(Beispiel 17) Sei ***CO1*** eine CO-Typdefinition im Entwurfsmodell im Namensraum ***M1***. Dann werden die folgenden CORBA-IDL-Definitionen erzeugt:

```
module M1 {
  interface CO1 : ComponentModel::ComponentBase {
    // ...
  };
```

```
    interface CO1COFactory : ComponentModel::CoFactoryBase {
      CO1 create_CO1 ();
    };
  };
```

2.1.8 Diskussion

Die Präsentation der konkreten Ableitungsregeln von $CORE_{MAP}$ für Definitionen der Struktursicht eines Entwurfsmodells auf Mechanismen von $CORE_{WARE}$ zeigt, daß die erzeugten CORBA-IDL-Definitionen außerordentlich komplex sein können. Dies ist insbesondere in den - notwendigerweise - verschiedenartigen Ableitungsregeln für die unterschiedlichen Interaktionsarten begründet. Durch das Konzept der Kombinationsmöglichkeit unterschiedlicher Interaktionsarten im Kontext eines Interfacetyps in $CORE_{CEPT}$ ist es jedoch gelungen, diese Komplexität nicht in der Entwurfsphase zu handhaben, sondern diese durch geeignete Ableitungsregeln von $CORE_{MAP}$ während der automatischen Ableitung von Softwarekomponenten zu behandeln.

Die Ableitungsregeln für die Definitionen der Struktursicht eines Entwurfsmodells erfüllen bereits einen Teil der in Abschnitt 1.1 formulierten Anforderungen. Der gemeinsame Kontext aller Interaktionsarten im Kontext eines Interfacetyps wird erhalten. Die Identität von COs wurde mittels der das CO repräsentierenden CORBA-IDL-Interfacedefinition realisiert, CO-Typ-abhängige und generische CO-Fabriken wurden ebenfalls definiert.

2.2 Abbildung der Konzepte der Konfigurationssicht

Die Konzepte der Konfigurationssicht sind - analog zu denen der Struktursicht - Bestandteile der externen Sicht auf einen CO-Typ. Der Abbildung der Strukturkonzepte folgend werden demzufolge CORBA-IDL-Konstrukte aus der Konfigurationssicht eines Entwurfsmodells erzeugt. Der Kerngedanke ist hier die Realisierung der Interface- bzw. CO-Navigation sowie des Reflexionsprinzips (vgl. Abschnitt 1.3). Der hier vorgestellte Ansatz folgt dem der Generierung von impliziten CORBA-IDL-Definitionen aus CORBA-Komponentendefinitionen [OMG CCM I]. Über [OMG CCMI] hinaus gehen die Konzepte dynamische *Port*-Definition sowie die mögliche Integration der drei Interaktionsarten im Kontext eines Interfacetyps im Entwurfsmodell. Die Ableitungsregeln für die mit CORBA-Komponenten semantisch vergleichbaren Konzepte des Konzeptraumes sind analog zu [OMGCCM I] definiert. Für die über das CORBA-Komponentenmodell hinausgehenden Konzepte wurden ebenfalls Ableitungsregeln in $CORE_{MAP}$ aufgenommen.

2.2.1 Provided-Port-Definition

Das Konzept *Port*-Definition dient dazu, Referenzen auf bereitgestellte Interfaces für die Umgebung eines COs, für das ein entsprechendes Entwurfsmodellelement definiert ist, verfügbar zu machen bzw. der Umgebung die Hinterlegung von Referenzen auf genutzte Interfaces zu gestatten. Das Konzept *Port*-Definition bezieht sich auf *Supports*- oder *Requires*-Relationen zwischen einem CO-Typ und einem Interfacetyp im Entwurfsmodell. Im Falle des Bezuges auf eine *Requires*-Relation können gerade Referenzen auf Interfaces hinterlegt werden, während bei *Supports*-Relationen Referenzen verfügbar gemacht werden.

Die Spezialisierung *Single-Port*-Definition schränkt das Konzept *Port*-Definition insofern ein, als das jeweils nur *eine* Referenz unter dem Namen der *Single-Port*-Definition hinterlegt bzw. verfügbar gemacht werden kann.

Für eine *Single-Port*-Definition im Kontext eines CO-Typs im Entwurfsmodell, die auf einer *Supports*-Relation basiert (*Provided-Port*-Definition), die zwischen dem CO-Typ und einem Interfacetyp definiert ist, werden in Abhängigkeit von den benutzten Interaktionsarten folgende Ableitungsregeln in $CORE_{MAP}$ definiert.

(Regel 26) Für jede Instanz des Konzeptes *Provided-Port*-Definition im Entwurfsmodell wird die CORBA-IDL-Interfacedefinition, die den definierenden CO-Typ repräsentiert, um eine Operation mit dem Namen **provide_<Port-Name>** erweitert. Der Rückgabetyp ist die CORBA-IDL-Repräsentation des Interfacetyps aus dem Entwurfsmodell, das der *Provided-Port*-Definition zugeordnet ist.

Die Anwendung von *Regel 26* erzeugt eine **provide**-Operation, die semantisch und syntaktisch der **provide**-Operation entspricht, die für eine CORBA-Komponente entsprechend [OMG CCM I] in der *Equivalent*-CORBA-IDL-Definition produziert wird. Es wird deutlich, daß die hier definierten Ableitungsregeln der Konfigurationssicht denjenigen des CORBA-Komponentenmodells für die - verglichen mit *CoRE* - vereinfachten CO-Typen (Meta-Typ *component* im CORBA-Komponentenmodell) semantisch entsprechen.

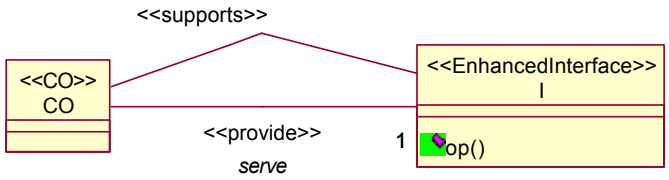


Abb. 11 Provided-Port-Definition

(Beispiel 18) Sei **CO** ein CO-Typ im Entwurfsmodell mit einer *Supports*-Relation zu einem Interfacetyp **I**. Auf der Basis der *Supports*-Relation sei eine *Provided-Port*-Definition mit dem Namen **serve** definiert. Dann werden entsprechend *Regel 26* die folgenden CORBA-IDL-Interfacedefinitionen produziert (illustriert in Abb.11):

```
module M1 {
  interface I {
  // Operationen und Attribute
  };

  interface CO : ::ComponentModel::ComponentBase {
    ::M1::I provide_serve ( );
  };
};
```

Der grundsätzliche Unterschied zwischen den Architekturen des CORBA-Komponentenmodells und dem hier vorgestellten Ansatz besteht in der Gleichbehandlung der verschiedenen Interaktionsarten im Kontext eines Interfacetyps, während das CORBA-Komponentenmodell Signalinteraktion und operationale Interaktion an *Port*-Definitionen unterscheidet. Zur Abbildung dieser kombinierten Interfacetypen standen für $CoRE_{MAP}$ zwei Abbildungsalternativen offen, die im folgenden erläutert werden sollen.

- *Veränderung der Signatur der in Regel 26 definierten* **provide**-*Operation*

  Entsprechend *Regel 11* bis *Regel 20* werden für Interfacetypen im Entwurfsmodell, die Signal- bzw. *Continuous-Media*-Interaktionselemente definieren, CORBA-IDL-Interfacedefinitionen produziert. In den CORBA-IDL-Definitionen für *Provided-Port*-Definition im Kontext derartiger Interfacetypen muß dementsprechend der Zugang zu CORBA-IDL-Interfacedefinitionen, die die Signal- bzw. *Continuous-Media*-Interaktionselemente abbilden, sichergestellt sein. Zu diesem Zweck kann durch $CoRE_{MAP}$ eine CORBA-IDL-**valuetype**-Definition erzeugt werden, die entsprechende *Public-Member*-Elemente mit dem Typ dieser CORBA-IDL-Interfacedefinitionen enthält. Diese CORBA-IDL-**valuetype**-Definition kann als Rückgabetyp der für eine *Provided-Port*-Definition produzierten **provide**-Operation definiert werden. Mit

Hilfe dieser Abbildungsmöglichkeit ist sicherlich - im Sinne der Erfordernisse von $CoRE_{WARE}$ - der Zugang zu allen Interaktionspunkten eines Interfacetyps möglich.

Jedoch hat die Veränderung der Signatur der definierten **provide**-Operation mehrere Nachteile. Zum einen kann nicht zwischen denjenigen CORBA-IDL-Interfacedefinitionen unterschieden werden, die für einen Klienten des bereitgestellten Interfaces von Relevanz sind (d.h., die für operationale Interaktionselemente produziert werden) und solchen, die $CoRE_{WARE}$ zur Realisierung der Signal- und *Continuous-Media*-Interaktion benötigt. Darüber hinaus erweist es sich als schwierig, mit diesem Ansatz sowohl typsichere als auch generische Mechanismen zum Erhalt von Referenzen auf CORBA-IDL-Interfaces zu realsieren. Klienten, die mit einem CO eines ihnen bekannten CO-Typs in Interaktion treten, benutzen sicherlich **provide**-Operationen, deren Rückgabetyp durch $CoRE_{MAP}$ produzierte CORBA-IDL-Interfacedefinitionen sind. Allgemeine Konfigurationswerkzeuge verwenden jedoch generische **provide**-Operationen zur Bereitstellung von Kommunikationskanälen für Signal- und *Continuous-Media*-Interaktionen, da diesen Werkzeugen die durch $CoRE_{MAP}$ produzierten CORBA-IDL-Definitionen i.allg. unbekannt sind. Letztendlich wird durch die Veränderung der Signatur der **provide**-Operation die semantische und syntaktische Äquivalenz zum CORBA-Komponentenmodell aufgehoben.

- *Ergänzung der Definition der CORBA-IDL-Interfacedefinition für einen CO-Typ im Entwurfsmodell um eine weitere Operation*

  Neben der CORBA-IDL-Definition der **provide**-Operation kann der CORBA-IDL-Interfacedefinition für einen CO-Typ eine weitere Operation hinzugefügt werden, die alle diejenigen CORBA-Interfacedefinitionen zusammenfaßt, die für Signal- oder *Continuous-Media*-Interaktionselemente produziert werden. Der Vorteil dieses Verfahrens besteht darin, daß zwischen den für einen Klienten und den für $CoRE_{WARE}$ relevanten CORBA-IDL-Interfacedefinitionen unterschieden werden kann. Darüber hinaus kann eine zusätzlich produzierte Operation Konfigurationsparameter aufnehmen, und zwar *ohne* Signaturveränderung der **provide**-Operation.

  Folgerichtig werden die Produktionsregeln für CO-Typdefinitionen in $CoRE_{MAP}$ ergänzt. Um alle CORBA-IDL-Interfacedefinitionen, die für Anbieter eines Interfaces mit Signal- oder *Continuous-Media*-Interaktionselementen generiert wurden, zusammenzufassen, wird entsprechend einer Regel eine CORBA-IDL-Interfacedefinition durch Spezialisierung dieser CORBA-IDL-Interfacedefinitionen produziert.

(Regel 27) Falls eine Instanz des Konzeptes Interfacetyp (Interfacetyp im Entwurfsmodell) eine oder mehrere Instanzen der Konzepte *Source*-, *Sink* oder *Consume*-Definition enthält, wird im Modul **<Interfacename>__Supply__** eine CORBA-IDL-Interfacedefinition mit dem Namen **<Interfacename>__Supply** erzeugt. Diese Interfacedefinition spezialisiert die CORBA-IDL-Interfacedefinitionen **<Interfacename>__Sinks**, **<Interfacename>__Sources** und **<Interfacename>__Consumer**, soweit diese entsprechend *Regel 15*, *Regel 17* bzw. *Regel 20* produziert wurden.

Durch die Definition von *Regel 27* wird $CoRE_{WARE}$ ein einheitlicher Zugang zu allen für nicht-operationale Interaktionsarten relevanten CORBA-IDL-Interfaces ermöglicht.

(Beispiel 19) Die in *Beispiel 18* dargestellte Situation sei um Signal- und *Continuous-Media*-Interaktionselemente erweitert. Es seien eine mit ***receive_Sig*** bezeichnete *Consume*-Definition zu dem in *Beispiel 4* definierten Signaltyp ***Sig1***, eine mit ***se*** bezeichnete *Source*-Definition und eine mit ***sk*** bezeichnete *Sink*-Definition basierend auf der im *Beispiel 12* eingeführten Medienmenge ***M*** definiert (vgl. Abb. 12). Dann werden durch *Regel 26* die folgenden CORBA-IDL-Definitionen impliziert:

```
module M1 {
  module I__Supply__ {
    interface sk__Sink {
      ::ComponentModel::MediaDevice sk_audio ( );
    };
```

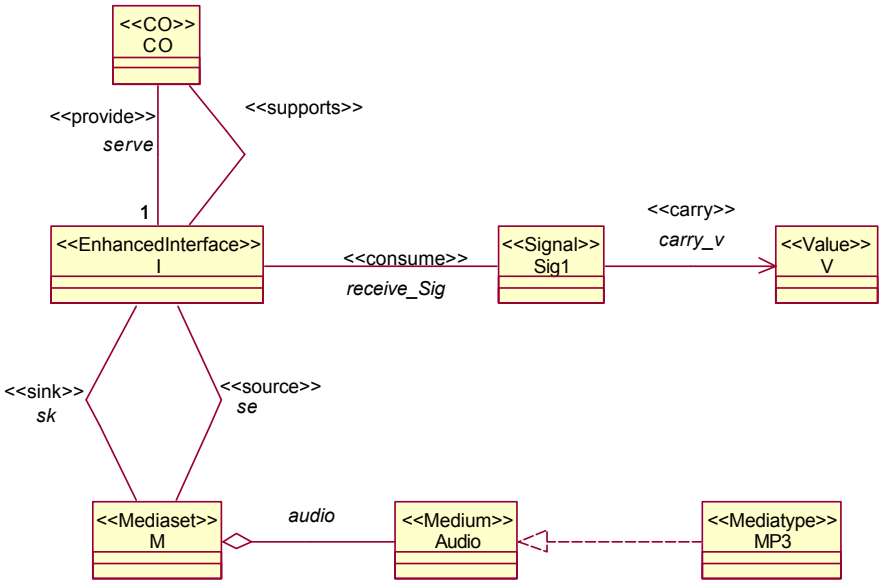


Abb. 12 Erweiterung der Interfacedefinition von *I* um Signal- und Continuous-Media-Interaktionselemente

```
      interface I__Sinks {
        readonly attribute ::M1::I__Supply__::sk__Sink sk;
      };

      interface se__Source {
        ::ComponentModel::MediaManager se_audio ( );
      };

      interface I__Sources {
        readonly attribute ::M1::I__Supply__::se__Source se;
      };

      interface receive_Sig_consumer : ::ComponentModel::ConsumerBase {
        void push_receive_Sig (
          in ::M1::Sig1 signal);
      };

      interface I__Consumer : receive_Sig_consumer {};

      interface I__Supply : I__Sinks, I__Sources, I__Consumer {};
    };
  };
```

Nun kann die Definition der zusätzlichen **provide**-Operation entsprechend folgender Regel vorgenommen werden.

(Regel 28) Für eine Instanz des Konzeptes *Provided-Port*-Definition zwischen einem CO-Typ und einem Interfacetyp im Entwurfsmodell wird die den CO-Typ repräsentierende CORBA-IDL-Interfacedefinition um eine Operation **provide_<Port-Name>_Supply** erweitert. Diese Operation hat einen **inout**-Parameter mit dem Namen **params** vom Typ **::ComponentModel::Parameters**. Der Rückgabetyp dieser Operation ist eine CORBA-IDL-Interfacedefinition, die für den Interfacetyp entsprechend *Regel 27* erzeugt wurde.

Dabei dient der **inout**-Parameter **params** der automatischen Konfiguration von Kommunikationskanälen für die Signal- und *Continuous-Media*-Interaktion durch $CORE_{WARE}$. Die Definition von **::ComponentModel::Parameters** wurde dabei generisch vorgenommen[1]:

```
module ComponentModel {
  typedef sequence<ValueBase> Parameters;
  };
```

Eine konkrete Ausprägung von Parametern sind Instanzen von CORBA-**valuetype**-Definitionen zur Identifikation von CORBA-*Event*- bzw. *Notification*-Kanälen zur Konfiguration von Signalkommunikationskanälen durch $CORE_{WARE}$:

```
module ComponentModel {
  module Container {
    valuetype ChannelIdentification {
      public string channel_id;
      public string element_name;
      public CosEventChannelAdmin::EventChannel channel;
      factory create
        ( in string channel_id,
          in string element_name,
          in CosEventChannelAdmin::EventChannel channel );
    };
  };
};
```

Dabei dient das *Member*-Element **channel_id** der Identifizierung eines Kommunikationskanals selbst. Um die Eindeutigkeit eine Kanalidentifikation zu sichern, wird ein global eindeutiger Name erzeugt (*Universal Unique ID* [MSCOM]). Das *Member*-Element **element_name** identifiziert ein Interaktionselement des Interfacetyps im Entwurfsmodell, also den Namen eines *Produce*- oder *Consume*-Interaktionselements. Das *Member*-Element **channel** referenziert das Administrationsinterface eines CORBA-*Event*- oder *Notification*-Kanals. Mit Hilfe dieses Mechanismus wird die Konfiguration verschiedenster Szenarien der Verknüpfung von Signalinteraktionspunkten von Interfaces ermöglicht: Jede Granularität dieser Konfiguration kann mittels **ChannelIdentification** ausgedrückt werden, d.h. ausgehend von der kleinsten Granularität (ein separater Kanal pro *Produce*- bzw. *Consume*-Interaktionselement) bis zur größten Granularität (ein Kanal realisiert alle Signalinteraktionen).

(Beispiel 20) Für die in *Beispiel 19* dargestellte Situation ergeben sich für den CO-Typ **CO**, der eine *Provided-Port*-Definition zu **I** definiert, die folgenden CORBA-IDL-Definitionen:

```
module M1 {
  interface CO : ::ComponentModel::ComponentBase {
    I provide_serve ( );
    I__Supply__::I__Supply provide_serve__Supply
      ( inout ::ComponentModel::Parameters params );
  };
};
```

Mit Hilfe der bisher eingeführten Regeln für die Konzepte der Konfigurationssicht können *Provided-Port*-Definitionen (für *Single-Port*-Definitionen) auf die externe Sicht auf einen CO-Typ in $CORE_{WARE}$ abgebildet werden. Es ist damit möglich, das Anbieten von Interfaces im Entwurfsmodell durch Operationen von CORBA-IDL-Interfacedefinitionen, die einen CO-Typ in der Plattform repräsentieren, nachzubilden. Dabei wurde insbesondere die Konfiguration von Kommunikationskanälen für die Signal- und *Continuous-Media*-Interaktionselemente durch eine entsprechende Operation berücksichtigt.

1. Es sei darauf hingewiesen, daß die konkrete Ausprägungen von *::ComponentModel::Parameters* ausschließlich durch die *Component-Support*-Plattform, d.h. transparent für die Implementierung der *Business-Logic*-Teile definiert und genutzt werden.Insofern kann - entgegen der allgemeinen Regel, für Anwendungsprogrammierung ausschließlich getypte Konfigurationsparameter zu verwenden - hier ein generischer Mechanismus definiert werden.

2.2.2 Used-Port-Definition

Im folgenden wird nun dargestellt, auf welche Weise Instanzen des Konzeptes *Used-Port*-Definition (*Single-Port*-Definitionen) durch $CORE_{MAP}$ abgebildet werden. Dabei wird - analog zur Betrachtung von *Provided-Port*-Definitionen - zunächst der Fall der Definition eines *Required*-Interfacetyps (Interfacetyp im Entwurfsmodell, zu dem von einem CO-Typ eine *Requires*-Relation definiert ist) mit ausschließlich operationaler Interaktion behandelt.

(Regel 29) Für jede Instanz des Konzeptes *Used-Port*-Definition zwischen einem CO-Typs und einem Interfacetyp im Entwurfsmodell wird die CORBA-IDL-Interfacedefinition, die den CO-Typ entsprechend *Regel 22* repräsentiert, um eine Operation mit dem Namen **connect_<Port-Name>** erweitert. Die Operation besitzt einen **in**-Parameter mit dem Namen **connect_<Port-Name>_p**. Der Typ dieses Parameters entspricht der CORBA-IDL-Repräsentation des Interfacetyps. Dieser Operation wird eine **raises**-Definition für die CORBA-IDL-Ausnahmedefinition **::ComponentModel::AlreadyConnected** hinzugefügt.

Die Ausnahme **AlreadyConnected** wird dann ausgelöst, wenn zur Ausführungszeit bereits eine CORBA-Objektreferenz mittels der **connect**-Operation registriert wurde. Sie ist folgendermaßen definiert:

```
module ComponentModel {
  exception AlreadyConnected {};
  };
```

Offensichtlich ist also zur Änderung einer bereits verbundenen CORBA-Objektreferenz eine weitere Operation notwendig, mit deren Hilfe die Beziehung einer *Single-Port*-Definition zu einer genutzten CORBA-Objektreferenz getrennt werden kann.

(Regel 30) Zu jeder entsprechend *Regel 29* erzeugten **connect_<Port-Name>**-Operation wird eine weitere Operation mit dem Namen **disconnect_<Port-Name>** in der gleichen CORBA-IDL-Interfacedefinition erzeugt. Diese Operation hat keinen Rückgabetyp und keine Parameter, ihr wird eine **raises**-Definition für die CORBA-IDL-Ausnahmedefinition **::ComponentModel::NotConnected** hinzugefügt.

Die Ausnahme **NotConnected** wird immer dann ausgelöst, wenn die Operation **disconnect_<Port-Name>** gerufen wird, obwohl zum Ausführungszeitpunkt keine CORBA-Objektreferenz mittels **connect_<Port-Name>** hinterlegt wurde. Sie ist - analog zu **AlreadyConnected** - wie folgt definiert:

```
module ComponentModel {
  exception NotConnected {};
  };
```

Die Anwendung von *Regel 29* und *Regel 30* illustriert das folgende Beispiel, dargestellt in Abb. 13.

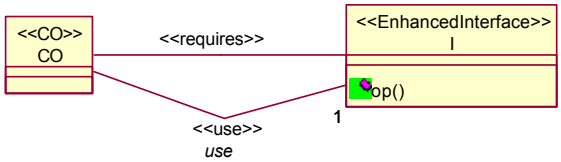


Abb. 13 Beispiel für eine Used-Port-Definition

(Beispiel 21) Sei ***I*** ein Interfacetyp im Namensraum ***M1*** eines Entwurfsmodells mit einer Operation ***op*** ohne Parameter und Rückgabetyp. Sei ***CO*** ein CO-Typ mit einer *Used-Port*-Definition für ***I*** mit dem Namen ***use***. Dann werden die folgenden CORBA-IDL-Definitionen produziert:

```
module M1 {
  interface I {
    void op ( );
  };
```

```
    interface CO : ::ComponentModel::ComponentBase {
      void connect_use ( in ::M2::I connect_use_p )
        raises ( ::ComponentModel::AlreadyConnected);
      void disconnect_use ( )
        raises ( ::ComponentModel::NotConnected );
    };
  };
```

Analog zur Darstellung von $CORE_{MAP}$ für *Provided-Port*-Definitionen muß im Falle von Signal- oder *Continuous-Media*-Interaktionselementen in genutzten Interfacetypen zu Konfigurationszwecken der Zugang zu CORBA-Objekten, die diese Interaktionselemente repräsentieren, ermöglicht werden. Wiederum standen für $CORE_{MAP}$ Alternativen zur Abbildung dieser Zugangsmechanismen offen:

- die Erweiterung der Signatur der entsprechend *Regel 29* produzierten **connect**-Operation bzw.
- die Bereitstellung einer weiteren Operation.

Im Falle einer weiteren Operation gestaltet es sich außerordentlich schwierig, die Korrelation zwischen dem CORBA-IDL-Interface, das den operationalen Anteil des Interfacetyps im Entwurfsmodell abbildet, und demjenigen, das dem Signal- und *Continuous-Media*-Interaktionsteil entspricht, zu realisieren. Insbesondere wären dann Szenarien denkbar, die in undefinierten Zuständen eines COs resultieren: Eine CORBA-Objektreferenz, die den operationalen Anteil des Interfacetyps im Entwurfsmodell referenziert, könnte mittels der **connect**-Operation hinterlegt werden, ohne daß der Signal- und *Continuous-Media*-Anteil initialisiert sein würde. Andererseits wäre in diesem Szenario die Repräsentation eines Interfacetyps im Entwurfsmodell als ganzheitlicher Interaktionskontext nicht gegeben.

Dementsprechend wurde in $CORE_{MAP}$ die Erweiterung der Signatur der **connect**-Operation realisiert. Dazu werden - analog zu *Regel 27* - diejenigen CORBA-IDL-Interfacedefinitionen, die für Nutzer eines Interfacetyps für Interaktionselemente *Source*, *Sink* und *Consume* produziert wurden, in einer CORBA-IDL-Interfacedefinition entsprechend der folgenden Regel zusammengefaßt.

(Regel 31) Falls eine Instanz des Konzeptes Interfacetyp im Entwurfsmodell eine oder mehrere Definitionen der Arten *Source*, *Sink* oder *Consume* definiert, wird im Modul **<Interfacename>__Use__** eine CORBA-IDL-Interfacedefinition mit dem Namen **<Interfacename>__Use** erzeugt. Diese Interfacedefinition spezialisiert die CORBA-IDL-Interfacedefinitionen **<Interfacename>__Sinks**, **<Interfacename>__Sources** und **<Interfacename>__Consumer**, soweit diese entsprechend *Regel 15*, *Regel 17* bzw. *Regel 20* produziert wurden.

Analog zu Signal- oder *Continuous-Media*-Interaktionselementen im Kontext einer *Provided-Port*-Definition, gestattet die CORBA-IDL-Interfacedefinition **<Interfacename>__Use** $CORE_{WARE}$ den direkten Zugriff auf alle CORBA-Objekte für Signal- oder *Continuous-Media*-Interaktionselemente. Die Anwendung von *Regel 31* verdeutlicht das folgende Beispiel.

(Beispiel 22) Für die in *Beispiel 19* dargestellte Situation ergeben sich für Nutzer des Interfacetyps **I** die folgenden CORBA-IDL-Definitionen:

```
module M1 {
  module I__Use__ {
    interface sk__Source {
      ::ComponentModel::MediaManager sk_audio ( );
    };
    interface I__Sources {
      readonly attribute ::M1::I__Use__::sk__Source sk;
    };
    interface se__Sink {
      ::ComponentModel::MediaDevice se_audio ( );
    };
    interface I__Sinks {
      readonly attribute ::M1::I__Use__::se__Sink se;
    };
```

```
    interface I__Use : I__Sources,  I__Sinks
    {};
  };
};
```

Mit Hilfe der durch die Anwendung von *Regel 31* produzierten CORBA-IDL-Interfacedefinition zur Zusammenfassung der CORBA-IDL-Interfacedefinitionen, die die Signal- und *Continuous-Media*-Interaktionselemente repräsentieren, kann nun die Signatur der **connect**-Operation entsprechend den Vorbetrachtungen erweitert werden.

(Regel 32) Die durch Anwendung von *Regel 29* erzeugte Operation **connect_<Port-Name>** wird um den Rückgabetyp **<Interfacename>__Use__::<Interfacename>__Use** erweitert. Zusätzlich erhält sie einen weiteren **inout**-Parameter mit dem Namen **params**. Der Typ dieses Parameters ist **::ComponentModel::Parameters**.

Analog zur Abbildung von *Provided-Port*-Definitionen werden Konfigurationsparameter mittels des Operationsparameters **params** übermittelt. Die durch $CORE_{MAP}$ erzeugte Realisierung der **connect**-Operation wird also das CORBA-IDL-Interface, das den operationalen Anteil eines Interfaces implementiert, als Parameter empfangen, die Konfiguration der Signal- und *Continuous-Media*-Kommunikationskanäle entsprechend den gelieferten Parametern vornehmen und die entsprechende CORBA-Objektreferenz als Rückgabewert zur weiteren Konfiguration durch $CORE_{WARE}$ liefern. Eine Anwendung von *Regel 32* verdeutlicht das folgende Beispiel (vgl. Abb.14).

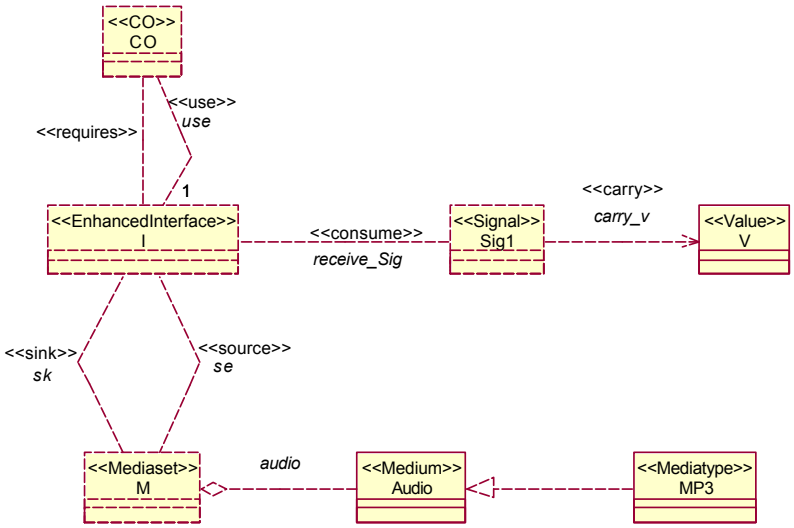


Abb. 14 Used-Port-Definition für einen Interfacetyp mit kombinierten Interaktionsarten

(Beispiel 23) Sei ***Client*** eine CO-Typdefinition im Entwurfsmodell, die eine *Used-Port*-Definition mit dem Namen ***use*** zum in *Beispiel 19* definierten Interfacetyp ***I*** spezifiziert. Dann ergeben sich durch Anwendung von *Regel 32* die folgenden CORBA-IDL-Definitionen:

```
module M1 {
  interface Client : ::ComponentModel::ComponentBase {
    ::M1::I__Use__::I__Use connect_use (
        in ::M1::I connect_use_p,
        inout ::ComponentModel::Parameters params)
      raises ( ::ComponentModel::AlreadyConnected );
  };
};
```

Mit den bisher vorgestellten Ableitungsregeln lassen sich Konfigurationen eines CO-Typs durch *Used-Port*- und *Provided-Port*-Definitionen mit den Mechanismen von $CORE_{MAP}$ und $CORE_{WARE}$ ausdrücken. Diese Konfigurationsart entspricht semantisch den Ausdrucksmitteln des *Port*-Konzeptes des CORBA-Komponentenmodells. *CORE* geht bezüglich Konfigurationsaspekten mit der Bereitstellung des Konzeptes *Multiple Port* über die Konzepte des CORBA-Komponentenmodells hinaus.

2.2.3 Multiple Port

Multiple-Port-Definitionen (d.h. Instanzen der Konzepte *Use*d- und *Provided-Port*-Definition mit der Auszeichnung *multiple*) gestatten die dynamische Anforderung bzw. Hinterlegung von Interfacereferenzen an COs. Der Hauptgrund für die Einführung von *Multiple-Port*-Definitionen liegt hier im erzielbaren höheren Maß an Skalierbarkeit: Für *Single-Port*-Definitionen kann die Realisierung der Interfacenavigation durch Ableitungsregeln von $CORE_{MAP}$ innerhalb des generierten Codes behandelt werden - diese Realisierung ist für die *Business-Logic*-Implementierung der Artefakte transparent. Für *Multiple-Port*-Definitionen wird durch Artefaktrealisierungen die Semantik der Bereitstellung von Interfacereferenzen implementiert. Gegenüber dem CORBA-Komponentenmodell ist mit diesen Ausdrucksmitteln eine wesentlich flexiblere Konfiguration von verteilten Softwaresystemen möglich: Die Anzahl von bereitzustellenden bzw. nutzbaren Interfacereferenzen muß nicht bereits in der Entwurfsphase bekannt sein. Die konkrete Semantik der dynamischen Bereitstellung von Interfacereferenzen ist Bestandteil der Artefaktrealisierung und wird in der Implementierungsphase definiert.

2.2.3.1 Provided-Port-Definition mit der Auszeichnung Multiple

Multiple-Port-Definitionen haben insbesondere eine Repräsentation in der externen Sicht auf einen CO-Typ, dementsprechend werden die CORBA-IDL-Definitionen, die einen CO-Typ repräsentieren, um entsprechende **provide**-Operationen für *Multiple-Port*-Definitionen erweitert. Dazu wird zunächst mittels der folgenden Ableitungsregel ein CORBA-IDL-Datentyp definiert, der die CORBA-IDL-Interfacedefinitionen für den operationalen Anteil eines Interfacetyps und den für die definierten Signal- bzw. *Continuous-Media*-Interaktionselemente erzeugten Anteil zusammenfaßt. Diese Zusammenfassung stellt zur Ausführungszeit die Korrelation zwischen den diese CORBA-IDL-Interfacedefinitionen repräsentierenden CORBA-Objektreferenzen in einem Kontext her.

(Regel 33) Für eine Instanz des Konzeptes Interfacetyp, die im Entwurfsmodell im Kontext einer *Provided-Port*-Definition mit der Auszeichnung *Multiple* auftritt, wird eine CORBA-IDL-**valuetype**-Datentypdefinition mit dem Namen **<Interfacename>_Access** im CORBA-IDL-Modul **<Interfacename>__Supply__** erzeugt. Diese CORBA-IDL-**valuetype**-Definition enthält ein Element mit dem Namen **operational_part**, das vom Typ der CORBA-IDL-Interfacedefinition ist, die den Interfacetyp im Entwurfsmodell repräsentiert. Falls für den Interfacetyp im Entwurfsmodell neben operationalen auch Signal- oder *Continuous-Media*-Interaktionselemente spezifiziert wurden, erhält die CORBA-IDL-**valuetype**-Definition ein weiteres Element des Namens **event_and_media_part** vom Typ **<Interfacename>__Supply**.

Eine Instanz dieser produzierten CORBA-IDL-**valuetype**-Definition wird durch eine CO-Typrealisierung benutzt, um zur dynamischen Anforderung einer Interfacereferenz die CORBA-Objektreferenzen der CORBA-IDL-Interfaces zu übergeben, die die operationalen und nicht-operationalen Interaktionselemente repräsentieren. Durch die Nutzung des CORBA-IDL-Datentyps **valuetype** kann die besondere Situation der Nichtverfügbarkeit von Interfacereferenzen mittels Rückgabe von **NULL** ausgedrückt werden.

Zur Verdeutlichung dieser Konzepte wird die Situation aus *Beispiel 19* um eine *Provided-Port*-Definition mit der Auszeichnung *Multiple* erweitert. Dies ist in Abb.15 dargestellt.

(Beispiel 24) Sei für den in *Beispiel 19* definierten CO-Typ ***CO*** eine *Provided-Port*-Definition mit dem Namen ***multiple_serve*** und der Auszeichnung *Multiple* zu ***I*** definiert. Dann werden folgende CORBA-IDL-Definitionen erzeugt:

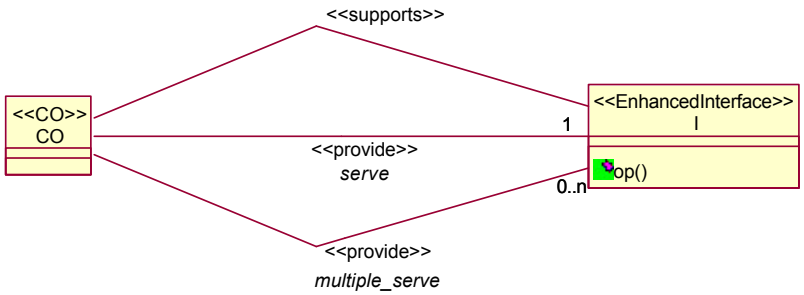


Abb. 15 Erweiterung der Definitionen von *Beispiel 19* um eine Multiple-Port-Definition

```
module M1 {
  module I__Supply__ {
    interface I__Supply;
    valuetype I_Access {
      public ::M1::I operational_part;
      public ::M1::I__Supply__::I__Supply event_and_media_part;
    };
  };
};
```

Die Bereitstellung von Interfacereferenzen erfolgt im Falle von *Multiple-Port*-Definitionen dynamisch, wobei die Semantik dieser Bereitstellung durch Implementierungen von Artefaktrepräsentationen durch den Entwickler in der Implementierungsphase realisiert wird. Dementsprechend resultiert die Definition einer *Multiple-Port*-Definition in der Produktion einer CORBA-IDL-Interfacedefinition, das die Umgebung eines COs zur dynamischen Interfacebereitstellung nutzt. Diese CORBA-IDL-Interfacedefinition wird von spezifischen Artefaktrealisierungen implementiert.

(Regel 34) Für eine Instanz des Konzeptes Interfacetyp im Entwurfsmodell, die im Kontext einer *Provided-Port*-Definition mit der Auszeichnung *Multiple* auftritt, wird eine CORBA-IDL-Interfacedefinition mit dem Namen **<Interfacename>__Dynamic** im Modul **<Interfacename>__Supply__** erzeugt. Diese CORBA-IDL-Interfacedefinition spezifiziert die Operation **aquire**. Diese Operation hat einen **in**-Parameter vom Typ **::ComponentModel::Parameters**. Der Rückgabetyp dieser Operation ist die entsprechend *Regel 33* produzierte CORBA-IDL-**valuetype**-Definition.

Der **in**-Parameter vom Typ **::ComponentModel::Parameters** wird durch $CORE_{WARE}$ wiederum zu Konfigurationszwecken genutzt. Konkrete derartige Konfigurationsparameter umfassen die Identifikation der zur Signal- und *Continuous-Media*-Interaktion notwendigen Mechanismen, die durch die $CORE_{WARE}$ bereitgestellt werden (vgl. Abschnitt 2.2.1, S. 45).

(Beispiel 25) Für die in *Beispiel 24* dargestellte Situation werden durch Anwendung von *Regel 34* die folgenden CORBA-IDL-Definitionen erzeugt:

```
module M1 {
  module I__Supply__ {
    interface I__Dynamic {
      ::M1::I__Supply__::I_Access acquire ( in ::ComponentModel::Parameters params );
    };
  };
};
```

Die durch $CORE_{MAP}$ erzeugten Repräsentationen von CO-Typen werden um eine Operation für *Multiple-Port*-Definitionen erweitert.

(Regel 35) Für eine Instanz des Konzeptes *Provided-Port*-Definition mit der Auszeichnung *Multiple* im Entwurfsmodell wird in der CORBA-IDL-Interfacedefinition, die den CO-Typ repräsentiert, eine Operation mit dem Namen **provide_<Multiple-Port-Name>** erzeugt. Diese erhält die entsprechend *Regel 34* erzeugte CORBA-IDL-Interfacedefinition **<Interfacename>__Dynamic** als Rückgabetyp.

(Beispiel 26) Für die in *Beispiel 24* präsentierte Situation (vgl. Abb.15) ergibt sich die folgende CORBA-IDL-Interfacedefinition für den CO-Typ ***CO***.

```
module M1 {
  interface CO : ::ComponentModel::ComponentBase {
    ::M1::I provide_serve ( ); // Single-Port-Definition serve, operationaler Teil

    // Single-Port-Definition serve, nicht-operationaler Teil
    ::M1::I__Supply__::I__Supply provide_serve__Supply ( inout ::ComponentModel::Parameters params);

    // Multiple-Port-Definition multiple_serve
    ::M1::I__Supply__::I__Dynamic provide_multiple_serve ( );
  };
};
```

2.2.3.2 Used-Port-Definition mit der Auszeichnung Multiple

Entsprechend den Darstellungen in Abschnitt 2.2.3.1 wurde für *Provided-Port*-Definitionen mit der Auszeichnung *Multiple* eine CORBA-IDL-**valuetype**-Definition benutzt, die den Kontext der CORBA-Objekte für den operationalen und den nicht-operationalen Teil des zugehörigen Interfacetyps herstellt. Diese Datentypdefinition wird als Rückgabetyp der entsprechenden **provide**-Operation verwendet. Die entsprechend *Regel 33* produzierte CORBA-IDL-**valuetype**-Definition kann verwendet werden, um an *Multiple-Port*-Definitionen Interfacereferenzen zu hinterlegen. Dieses Vorgehen hätte eine Erweiterung der CORBA-IDL-Interfacedefinitionen zur Folge, die einen CO-Typ mit *Used-Port*-Definition mit der Auszeichnung *Multiple* repräsentieren. Diese Abbildung ist im Falle einer *Provided-Port*-Definition wegen der Korrelation von operationalem und nicht-operationalem Anteil notwendig. Im Fall einer *Used-Port*-Definition ist dagegen diejenige Abbildung sinnvoll, die die Definition der den CO-Typ repräsentierenden CORBA-IDL-Interfacedefinition nicht erweitert. Diese Forderung ist legitim, da die in dieser CORBA-IDL-Interfacedefinition enthaltene Signatur der **connect**-Operation bereits die Korrelation zwischen operationalem und nicht-operationalem Anteil sichert (vgl. Abschnitt 2.2.2). Darüber hinaus ist es wünschenswert, die Definition der Ableitungsregeln so weit wie möglich den Ableitungsregeln anzupassen, die durch das CORBA-Komponentenmodell definiert werden. Diesen Grundsätzen entsprechend wurde die folgende Ableitungsregel in $CORE_{MAP}$ definiert.

(Regel 36) Für eine Instanz des Konzeptes *Used-Port*-Definition mit der Auszeichnung *Multiple* im Kontext eines CO-Typs im Entwurfsmodell wird die entsprechend *Regel 29* bzw. *Regel 32* erzeugte **connect**-Operation insofern in ihrer Signatur verändert, als das das potentielle Auslösen der Ausnahme **AlreadyConnected** entfernt wird.

Die Definition dieser Regel entspricht der geforderten Beibehaltung der CORBA-IDL-Interfacedefinition, die den CO-Typ repräsentiert, der die *Multiple-Port*-Definition enthält. Im Gegensatz zum *Single-Port*-Fall kann diese Operation nun mehrmals ohne vorherige Aufhebung der Bindung der *Port*-Definition an eine Interfacereferenz gerufen werden.

(Beispiel 27) Die in *Beispiel 24* dargestellte Situation sei um die Definition eines CO-Typs ***AClient*** erweitert., der eine *Requires*-Relation zum Interfacetyp ***I*** definiert. Auf dieser Relation basierend wird die *Multiple-Port*-Definition ***multiple_use*** spezifiziert. Dann ergeben sich durch Anwendung von *Regel 36* die folgenden CORBA-IDL-Definitionen.

```
module M1 {
  interface AClient : ::ComponentModel::ComponentBase {
    ::M1::I__Use__::I__Use connect_multiple_use ( in ::M1::I connect_multiple_use_p,
      inout ::ComponentModel::Parameters params);
  };
};
```

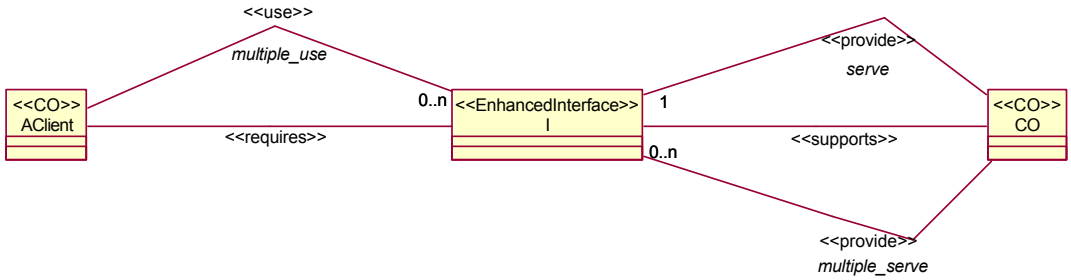


Abb. 16 Beispiel einer Multiple-Port-Definition für eine Requires-Relation

2.2.4 Diskussion

Die Darstellung der Ableitungsregeln für die Konzepte der Konfigurationssicht von *CORE* vervollständigt die Repräsentation der Kombinationsmöglichkeit unterschiedlicher Interaktionsarten im Kontext eines Interfacetyps im Entwurfsmodell innerhalb von $CORE_{WARE}$. Des weiteren wird plausibel gemacht, daß das Konzept der dynamischen Bereitstellung oder Hinterlegung von Interfacereferenzen durch Ableitungsregeln von $CORE_{MAP}$ umsetzbar ist. Insofern stellt der hier vorgestellte Abbildungsansatz auf Komponentenarchitekturen eine wesentliche Erweiterung des CORBA-Komponentenmodells [OMGCCMI] dar, das nur statische Konfigurationsmöglichkeiten für den Austausch von Interfacereferenzen bietet.

Die Darstellung der Ableitungsregeln für die Konzepte der Implementierungssicht von *CORE* wird in ihrer Kombination mit den hier vorgestellten Regeln für die Konfigurationssicht zeigen, daß das Management von *Port*-Definitionen ausschließlich durch entsprechend Ableitungsregeln von $CORE_{MAP}$ produzierten Implementierungscode erfolgen kann. Dieser Implementierungscode realisiert diejenigen Verhaltensteile eines CO-Typs, die dem Konfigurationsmanagement zugeordnet werden können. Einzige Ausnahme bildet dabei die Implementierung des spezifischen Verhaltens während der Beschaffung einer dynamisch erzeugten Interfacereferenz im Kontext einer *Provided-Port*-Definition mit der Auszeichnung *Multiple*.

Gemeinsam mit den Ableitungsregeln der Struktursicht werden durch die Regeln der Konfigurationssicht CORBA-IDL-Definitionen erzeugt, die die im Entwurfsmodell definierte externe Sicht auf einen CO-Typ strukturäquivalent in $CORE_{WARE}$ abbilden. Durch $CORE_{MAP}$ entsteht für solche Entwurfsmodelle, die auf den Konzepten des CORBA-Komponentenmodells basieren, eine den Ableitungsregeln für CORBA-Komponentendefinitionen aus [OMG CCMI] semantisch äquivalente Menge von CORBA-IDL-Definitionen.

2.3 Abbildung der Konzepte der Implementierungssicht

Die konkreten Ableitungsregeln von $CORE_{MAP}$ für die Konzepte der Implementierungssicht von *CORE* hängen zum einen von der zugrundeliegenden *Distributed-Processing*-Umgebung (im Falle von $CORE_{WARE}$ ist das CORBA 2.4) als auch der verwendeten Programmiersprache für die Implementierung ab. In *CORE* wurde exemplarisch C++ gewählt, die Prinzipien der Ableitungsregeln sind jedoch auf andere Programmiersprachen übertragbar.

Die primäre Zielstellung bei der Abbildung der Konzepte der Implementierungssicht ist der Aufbau einer programmiersprachlichen Repräsentation der das Verhalten eines CO-Typs realisierenden Artefakte und deren Integration mit den Repräsentationen der Interaktionselemente. Grundlage dieser Integration sind die Ableitungsregeln, die in $CORE_{MAP}$ für die Konzepte der Struktur- und Konfigurationssicht definiert wurden.

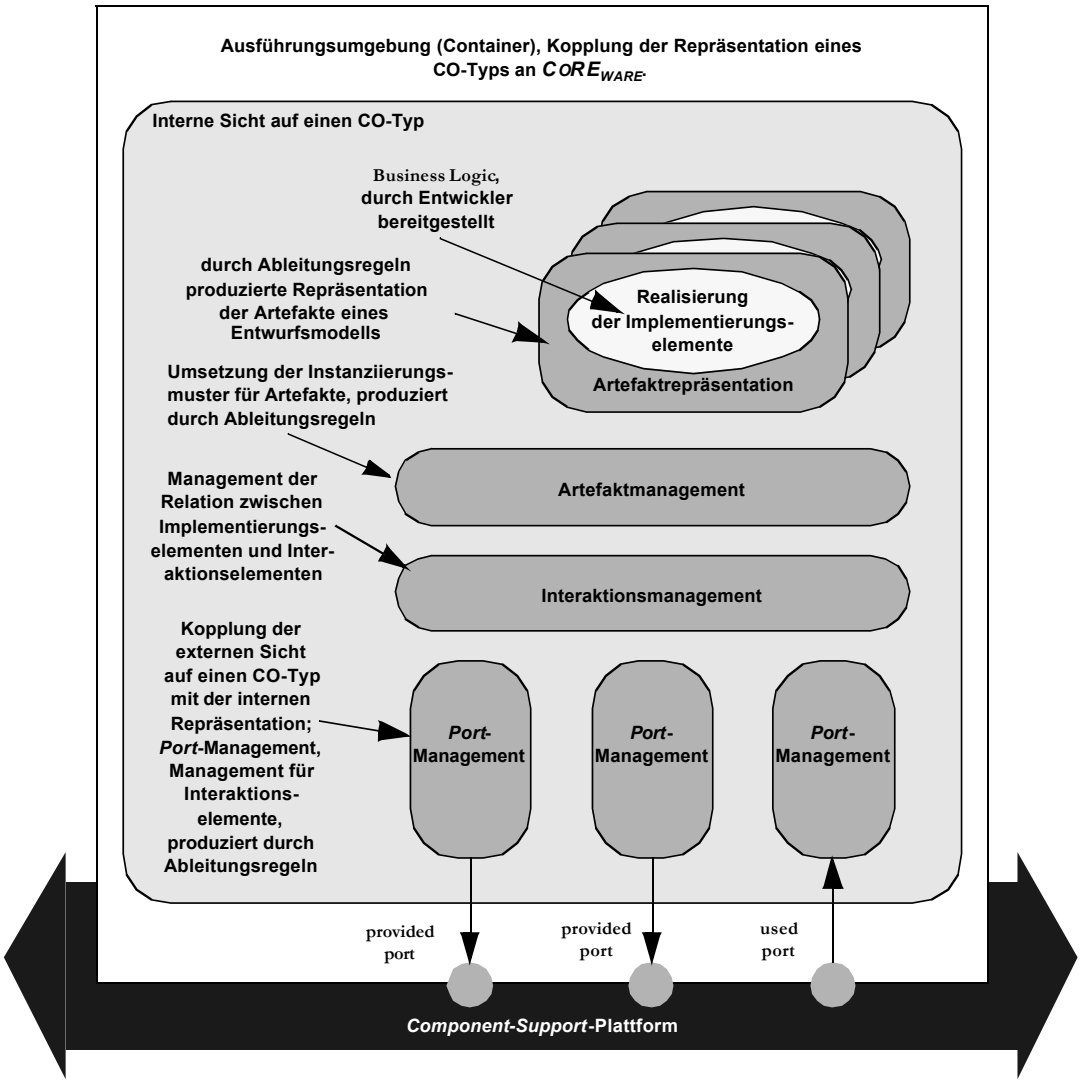


Abb. 17 Interne Sicht auf einen CO-Typ

Eine schematische Übersicht über die Bestandteile der internen Sicht auf einen CO-Typ liefert Abb. 17. Die generelle Aufgabe dieser Bestandteile besteht in der Kopplung der externen Sicht auf einen CO-Typ (erzeugt durch Anwendung von $CORE_{MAP}$ auf Instanzen der Konzepte der Struktur- und Konfigurationssicht) mit den durch den Entwickler bereitgestellten Realisierungen der Implementierungselemente der Artefaktrepräsentationen. Die Artefaktrepräsentationen werden durch Ableitungsregeln aus den Entwurfsmodellelementen erzeugt. In der gewählten objektorientierten Programmiersprache C++ werden Artefaktdefinitionen kanonisch durch C++-Klassen repräsentiert. Für die Artefakten im Entwurfsmodell zugewiesenen Implementierungselemente werden Klassenmethoden dieser C++-Klassen erzeugt. Die im Entwurfsmodell definierten Instanziierungsregeln werden auf artefaktspezifische *Factory*-Klassen abgebildet, die Instanzen der Artefaktrepräsentationen erzeugen können. Diese *Factory*-Klassen werden durch das Artefaktmanagement

als ein zentraler Bestandteil der internen Sicht auf einen CO-Typ zur Anforderung von Artefaktinstanzen für Interaktionen genutzt. Die Bestandteile des Interaktionsmanagement stellen die Korrelation zwischen Interaktionselementen von Interfacetypen und Implementierungselementen von Artefakten her. Das *Port*-Management koppelt das Interaktionsmanagement an die externe Sicht auf CO-Typen.

Bereits der schematischen Darstellung der internen Sicht auf CO-Typen kann entnommen werden, daß die aus den Ableitungsregeln und der entsprechenden Codegenerierung resultierenden Bestandteile des *Port*-, Interaktions- und Artefaktmanagement einen bedeutenden Teil der Verhaltensrealisierung von CO-Typen bilden. Insbesondere ist jede Abhängigkeit von den Konstrukten der *Component-Support*-Plattform und der ihr zugrundeliegenden *Distributed-Processing*-Umgebung in diesen Ableitungsregeln gekapselt. Die Ableitungsregeln von $CORE_{MAP}$ ermöglichen die Generierung von - bis auf die Realisierung der Implementierungselemente - vollständigen Repräsentation von CO-Typen in $CORE_{WARE}$.

Wie in *Anforderung (17) in Abschnitt 1.1* gefordert, müssen alle Konzepte des Konzeptraumes auf entsprechende Mechanismen der *Component-Support*-Plattform bzw. der *Distributed-Processing*-Umgebung von $CORE_{WARE}$ abbildbar sein. Diese Anforderung trifft insbesondere auch auf die im Entwurfsmodell definierte Implementierungsstruktur zu. Insofern erweitert $CORE_{MAP}$ das CORBA-Komponentenmodell in Bezug auf eine vollständige programmiersprachliche Abbildung der Definitionen der Implementierungssicht eines Entwurfsmodells, deren Äquivalent eine *Component-Implementation-Definition-Language*-Spezifikation (CIDL) im CORBA-Komponentenmodell [OMG CCM I] darstellt. Die Ableitungsregeln für CIDL-Spezifikationen in Programmiersprachen wurde jedoch bisher durch OMG nicht vorgenommen. In einem gemeinsamen Forschungsprojekt zwischen EURESCOM und der Humboldt-Universität zu Berlin [CCMP00] wurde nachgewiesen, daß die Konzepte der konkreten Ableitungsregeln für die Implementierungssicht auch für eine programmiersprachliche Abbildung von CIDL einsetzbar sind.

Die Diskussion der Ableitungsregeln für die Konzepte der Implementierungssicht erfolgt im weiteren anhand des folgenden Beispiels:

(Beispiel 28) Sei **O** eine Instanz des Konzeptes CO-Typ im Entwurfsmodell im Namensraum **M1**. Zwischen **O** und einer Instanz **I** des Konzeptes Interfacetyp im Entwurfsmodell im Namensraum **M1** seien eine *Requires*-Relation sowie eine *Supports*-Relation definiert. Für **I** sei eine parameterlose Operation **op** ohne Rückgabetyp definiert. Weiterhin seien eine mit **sendSig** bezeichnete *Produce*-Definition und eine mit **receiveSig** bezeichnete *Consume*-Definition im Kontext von **I**, basierend auf dem Signaltyp **Sig** definiert. Der CO-Typ **O** besitzt eine *Used-Port*-Definition **use** zu **I** ohne die Auszeichnung *Multiple* sowie eine *Provided-Port*-Definition basierend auf **I** des Namens **serve** ohne die Auszeichnung *Multiple*. Diese Situation ist in Abb. 18 dargestellt.

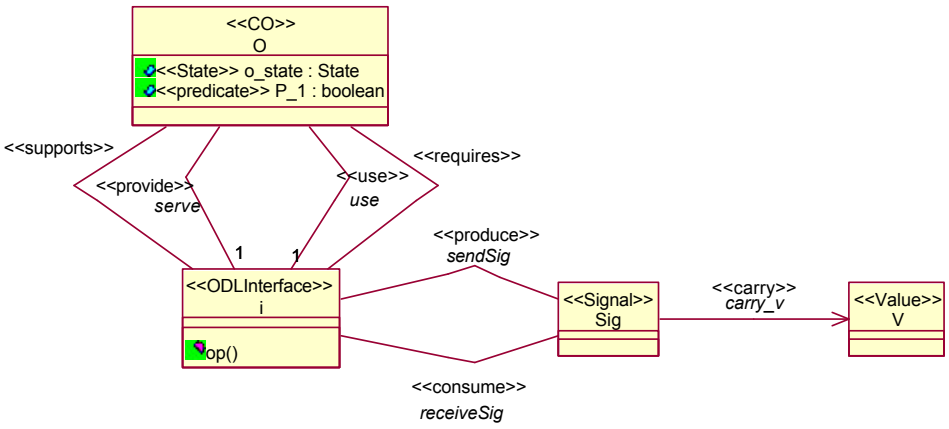


Abb. 18 Beispiel für Ableitungsregeln der Definitionen der Implementierungssicht

2.3.1 Repräsentation der externen Sicht auf ein CO - Port-Management

Alle C++-Definitionen, die durch $CoRE_{MAP}$ erzeugt werden, erfolgen in C++-Namensräumen (*Name Spaces*). Diese Namensräume korrespondieren zu den CORBA-IDL-Modulen, die durch die Ableitungsregeln entsprechend Abschnitt 2.1 und Abschnitt 2.2 definiert sind.

(Regel 37) Für jedes entsprechend *Regel 1* und *Regel 10* produzierte CORBA-IDL-Modul wird ein gleichnamiger C++-Namensraum erzeugt. Verschachtelungen derartiger CORBA-IDL-Module werden auf verschachtelte C++-Namensräume abgebildet.

In der CORBA-basierten Komponentenarchitektur von $CoRE_{WARE}$ werden Operationen, die durch CORBA-IDL-Interfacedefinitionen spezifiziert sind, durch *Servant*-Konstrukte (i.allg. Klassen einer Programmiersprache) implementiert. Die Identität sowie Lebenszeit von CORBA-Objekten und instanziierten *Servant*-Konstrukten, die die Operationen der von diesen CORBA-Objekten unterstützten CORBA-IDL-Interfacedefinitionen implementieren, ist voneinander unabhängig.

Bei der Kopplung der CO-Typrealisierung an $CoRE_{WARE}$ werden *Servant*-Klassen der Programmiersprache C++ für alle CORBA-IDL-Interfacedefinitionen produziert, die durch die Ableitungsregeln von $CoRE_{MAP}$ für Instanzen der Konzepte der Struktur- und Konfigurationssicht erzeugt werden. Die *Servant*-Klassen beinhalten Klassenmethoden, deren Implementierung die Operationen dieser CORBA-IDL-Interfaces realisieren. Die Definition der *Servant*-Klassen ergibt sich durch Anwendung der C++-*Language-Mapping*-Regeln [OMG C++ Map] auf die CORBA-IDL-Definitionen, die für die Instanzen der Konzepte der Struktur- und Konfigurationssicht in einem Entwurfsmodell durch $CoRE_{MAP}$ produziert werden. Die durch $CoRE_{MAP}$ erzeugte Implementierung dieser Klassenmethoden stellt also die Implementierung der Interaktionselemente im Kontext eines Interfacetyps im Entwurfsmodell bereit. Instanzen der erzeugten *Servant*-Klassen stellen die Anbindung der Repräsentationen der *Port*-Definitionen eines CO-Typs an die *Component-Support*-Plattform $CoRE_{WARE}$ her. Die durch $CoRE_{MAP}$ erzeugte Implementierung der *Servant*-Klassenmethoden benutzen das Entwurfsmuster Delegation [GHJ+ 99], um die Implementierung der Interaktionselemente durch Implementierungselemente der Artefaktrepräsentationen zu rufen. Diese Korrelation wird durch C++-Klassen (hier als *Composition*-Klassen bezeichnet) implementiert, die das Interaktionsmanagement realisieren.

2.3.1.1 Produktion von Servant-Klassen

Servant-Klassen werden für die CORBA-IDL-Interfacedefinitionen, die entsprechend den in Abschnitt 2.1 und Abschnitt 2.2 vorgestellten Ableitungsregeln produziert wurden, erzeugt. Die Zusammenhänge zwischen CORBA-IDL-Interfacedefinitionen und produzierten C++-*Servant*-Klassen stellt Abb. 19 dar.

Die Ableitungsregeln von $CoRE_{MAP}$ unterscheiden zwischen den CORBA-IDL-Interfacedefinitionen, die für den operationalen Teil eines Interfacetyps im Entwurfsmodell produziert werden, und jenen, die für Signal- und *Continuous-Media*-Interaktionselemente erzeugt werden.

(Regel 38) Für die CORBA-IDL-Interfacedefinition, die für den operationalen Anteil eine Instanz des Konzeptes Interfacetyp im Entwurfsmodell entsprechend *Regel 9* produziert wurde, wird eine korrespondierende *Servant*-Klasse erzeugt. Diese Klasse definiert Klassenmethoden, deren Signatur durch Anwendung der CORBA-*Language-Mapping*-Regeln für C++ [OMG C++Map] auf die CORBA-IDL-Interfacedefinition vorgegeben ist. Die *Servant*-Klasse als Spezialisierung (*Public Virtual*) der durch Anwendung von [OMG C++ Map] erzeugten POA-Klasse definiert. Falls die *Servant*-Klasse aus einer CORBA-IDL-Interfacedefinition erzeugt wird, die eine andere CORBA-IDL-Interfacedefinition spezialisiert (d.h. eine Generalisierungsrelation ist für den Interfacetyp im Entwurfsmodell definiert), so spezialisiert die *Servant*-Klasse kanonisch die die CORBA-IDL-Basisinterfacedefinition repräsentierende *Servant*-Klasse.
Der Konstruktor der *Servant*-Klasse erhält einen Parameter vom Typ **`const class <Interfacename>Composition *`**, der während der Konstruktion im *Private-Member*-Element **`_delegator`**

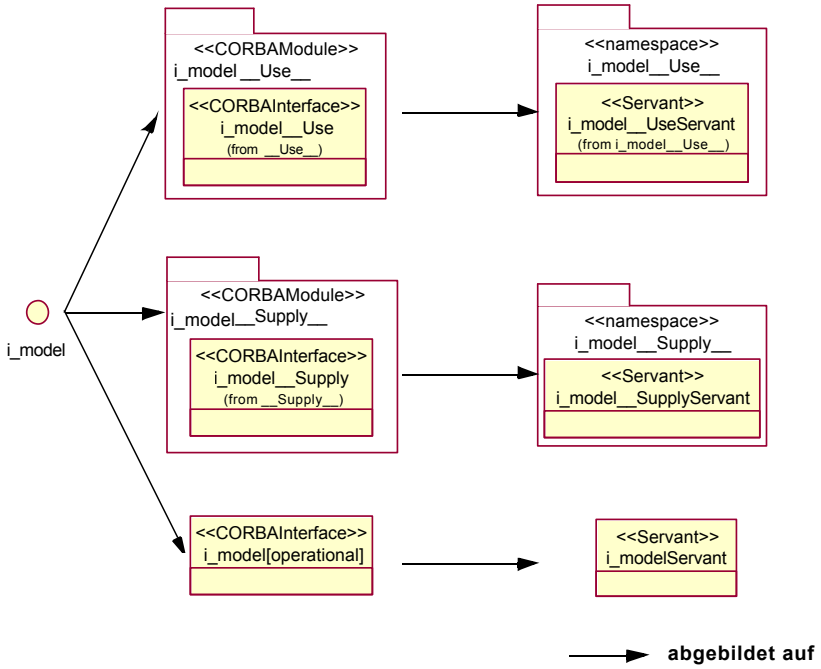


Abb. 19 Abbildung von CORBA-IDL-Interfacedefinitionen auf Servant-Klassen

hinterlegt wird, und auf den mittels der produzierten Klassenmethode **`get_delegator`** zugegriffen werden kann.

Der Parameter **`delegator`** des Konstruktors verweist auf eine *Composition*-Klasse des Interaktionsmanagement, an die die Implementierung der *Servant*-Klassenmethoden delegiert.

(Beispiel 29) Entsprechend *Regel 38* wird für den Interfacetyp ***i*** aus *Beispiel 28* die folgende C++-Klasse erzeugt:

```
namespace M1 {
 class iServant : public virtual POA_M1::i {
  const ::M1::iComposition * _delegator;
 public:
  iServant ( const ::M1::iComposition * delegator );
  ::M1::iComposition * get_delegator () const;

  virtual void op ();
 };
};
```

Analoge Ableitungsregeln können auch zur Produktion derjenigen *Servant*-Klassen eingesetzt werden, die die nicht-operationalen Interaktionselemente realisieren.

(Regel 39) Falls eine Instanz des Konzeptes Interfacetyp im Entwurfsmodell Interaktionselemente der Typen *Source*, *Sink*, *Produce* bzw. *Consume* definiert, wird *Regel 38* auch auf die CORBA-IDL-Interfacedefinitionen **<Interfacename>__Supply** bzw. **<Interfacename>__Use** (vgl. *Regel 27* bzw. *Regel 31*) sowie auf diejenigen CORBA-IDL-Interfacedefinitionen, die Basisinterfacedefinitionen von **<Interfacename>__Supply** bzw. **<Interfacename>__Use** sind, angewendet.

(Beispiel 30) Für *Beispiel 28* ergeben sich damit die folgenden *Servant*-Klassen für Anbieter des Interfacetyps ***i*** (*Supports*-Relation). Für Nutzer des Interfacetyps ***i*** (*Requires*-Relation) werden analoge C++-*Servant*-Klassen erzeugt[1].

```
namespace M1 {
 namespace i__Supply__ {
  class receiveSig_consumerServant
  : public virtual POA_M1::i__Supply__::receiveSig_consumer {
    const ::M1::iComposition * _delegator;
  public:
    receiveSig_consumerServant ( const ::M1::iComposition * delegator );
    ::M1::iComposition * get_delegator () const;

    virtual void push_receiveSig ( ::M1::Sig * signal );
    virtual void push ( const CORBA::Any& any);
    virtual void disconnect_push_consumer();
  };

  class i__ConsumerServant
  : public virtual POA_M1::i__Supply__::i__Consumer,
    public ::M1::i__Supply__::receiveSig_consumerServant
  {
    const ::M1::iComposition * _delegator;
  public:
    i__ConsumerServant ( const ::M1::iComposition * delegator );
    ::M1::iComposition * get_delegator () const;

    virtual void push ( const CORBA::Any& any);
    virtual void disconnect_push_consumer();
  };

  class i__SupplyServant
  : public virtual POA_M1::i__Supply__::i__Supply,
    public ::M1::i__Supply__::i__ConsumerServant
  {
    const ::M1::iComposition * _delegator;
  public:
    i__SupplyServant ( const ::M1::iComposition * delegator );
    ::M1::iComposition * get_delegator () const;
  };
 }
}
```

CO-Typen selbst werden entsprechend *Regel 22* ebenfalls durch CORBA-IDL-Interfacedefinitionen repräsentiert. Im Kontext dieser Interfacedefinitionen werden Operationen für *Provided*- bzw. *Used-Port*-Definitionen erzeugt. Für diese CORBA-IDL-Interfacedefinitionen werden durch Anwendung von *Regel 38* ebenfalls *Servant*-Klassen produziert. Dieses Prinzip wird in einer Ableitungsregel formuliert, die für die Interfacenavigation eines CO-Typs die entsprechenden *Servant*-Klassen erzeugt.

(Regel 40) Regel 38 wird auf die CORBA-IDL-Interfacedefinition, die einen CO-Typ entsprechend *Regel 22* in $CORE_{WARE}$ repräsentiert, angewendet. Anstelle des Parameters vom Typ **`const <Interfacename>Composition *`** wird nun **`const <CO-Typ>Composition *`** verwendet.

(Beispiel 31) Für die CORBA-IDL-Interfacedefinition **`::M1::O`**, die für den CO-Typ ***O*** aus *Beispiel 28* erzeugt wird, wird die folgende C++-*Servant*-Klasse erzeugt.

```
namespace M1 {
 class OServant : public virtual POA_M1::O {
   const ::M1::OComposition * _delegator;
 public:
   OServant ( const ::M1::OComposition * delegator );
   ::M1::OComposition * get_delegator () const;
```

1. Es sei darauf hingewiesen, daß die Klassenmethoden **`push`** und **`disconnect_push_consumer`** aus der Definition der einem Signalempfänger zugrundeliegenden CORBA-IDL-Interfacedefinition **`ConsumerBase`** als Spezialisierung von **`CosEventComm::EventConsumer`** resultieren.

```
    virtual ::M1::i_ptr provide_serve ();
    virtual ::M1::i__Supply__::i__Supply_ptr provide_serve__Supply
       ( ::ComponentModel::Parameters& params );
    virtual ::M1::i__Use__::i__Use_ptr connect_use
       ( ::M1::i_ptr connect_use_p, ::ComponentModel::Parameters& params );
  };
}
```

Für CORBA-IDL-Interfacedefinitionen, die für Interaktionselemente von Interfacetypen im Entwurfsmodell erzeugt wurden, werden als Ziel der Delegierung für *Servant*-Klassen C++-Klassen vom Typ **<Interfacename>Composition** (Interface-*Composition*-Klassen) generiert, die die Korrelation zwischen Interaktionselementen und Implementierungselementen in Artefaktrepräsentationen herstellen. Für einen CO-Typ selbst werden zu diesem Zweck C++-Klassen vom Typ **<CO-Typ>Composition** (CO-Typ-*Composition*-Klassen) erzeugt. Der genaue Zusammenhang zwischen diesen *Composition*-Klassen ist Bestandteil der nachfolgenden Darstellungen.

2.3.1.2 Interface- und CO-Typ-Composition-Klassen

Für jeden CO-Typ, der eine *Provided-Port*-Definition basierend auf einem Interfacetyp ***I*** und eine *Used-Port*-Definition zu einem Interfacetyp ***J*** spezifiziert, werden in seiner Repräsentation in der *Component-Support*-Plattform zunächst Instanzen von *Servant*-Klassen erzeugt. Für eine *Provided-Port*-Definition werden *Servant*-

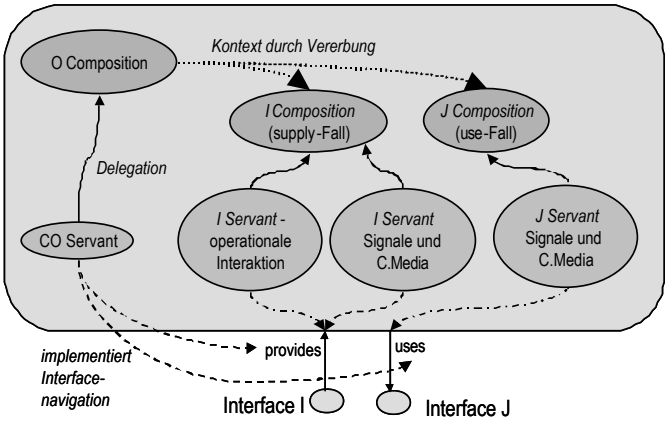


Abb. 20 Composition- und Servant-Klassen

Klassen jeweils für den operationalen und nicht-operationalen Anteil instanziiert, für eine *Used-Port*-Definition nur die *Servant*-Klasse für den nicht-operationalen Anteil. Die Implementierungen der Klassenmethoden dieser generierten *Servant*-Klassen delegieren an die Interface-*Composition*-Klassen. Zusätzlich wird eine Instanz derjenigen *Servant*-Klasse instanziiert, die für den CO-Typ selbst erzeugt wurde. Diese *Servant*-Klasse delegiert an die CO-Typ-*Composition*-Klasse. Alle *Servant*-Klasseninstanzen werden durch die Instanz der generierten CO-Typ-*Composition*-Klasse erzeugt, die somit für das *Port*-Management verantwortlich ist. Die Korrelation zwischen *Port*-Management und Interaktionsmanagement wird dadurch sichergestellt, daß die CO-Typ-*Composition*-Klasse von denjenigen Interface-*Composition*-Klassen ableitet, für die der CO-Typ im Entwurfsmodell *Provided-* bzw. *Used-Port*-Definitionen enthält. Damit ist den *Servant*-Klassen für angebotene bzw. genutzte Interfacetypen zwar nur der Typ der zugehörigen Interface-*Composition*-Klasse bekannt, zur Konstruktionszeit wird ihnen allerdings eine Instanz der CO-Typ-*Composition*-Klasse übergeben. Dadurch ist gesichert, daß der gemeinsame Kontext eines COs durch eine Instanz der CO-Typ-*Composition*-Klasse hergestellt wird. Die generierte Implementierung dieser Klasse instanziiert nachfolgend *Servant*-Klassen für jede *Provided-* bzw. *Used-Port*-Definition im Kontext des CO-Typs. Einen Überblick über die in *CORE* realisierte Lösung zum Management der Korrelation von Interaktionselementen und *Port*-Management liefert Abb. 20.

Entsprechend den Vorbetrachtungen können nun die Regeln zur Erzeugung der Interface-*Composition*- bzw. CO-Typ-*Composition*-Klassen formuliert werden.

(Regel 41) Für jede Instanz des Konzeptes Interfacetyp im Entwurfsmodell wird eine korrespondierende C++-Klasse mit dem Namen **<Interfacename>Composition** in einem C++-Namensraum erzeugt. Dieser Namensraum korrespondiert zu dem Namensraum im Entwurfsmodell, in dem der Interfacetyp definiert ist. Operationsrufe, die an einem durch eine *Servant*-Klasse implementierten CORBA-Interface eintreffen, werden an die korrespondierende **<Interfacename>Composition**-Klasse delegiert.

Während für alle CORBA-IDL-Interfacedefinitionen, die die Interaktionselemente eines Interfacetyps im Entwurfsmodell widerspiegeln, *Servant*-Klassen erzeugt wurden, aggregiert die Interface-*Composition*-Klasse diese *Servant*-Klassen. Sie stellt also den gemeinsamen Interaktionskontext im Rahmen der Implementierung her.

Die Definition dieser Interface-*Composition*-Klasse ergibt sich kanonisch aus der Definition der Methoden der *Servant*-Klassen, die an diese Klasse delegieren.

(Regel 42) Eine entsprechend *Regel 41* produzierte Interface-*Composition*-Klasse definiert zwei lokale Klassen

- **<Interfacename>Composition::__Supplier** und
- **<Interfacename>Composition::__User**.

Die lokale Klasse **__Supplier** enthält Methoden für alle Operationen der CORBA-IDL-Interfacedefinitionen des CORBA-IDL-Moduls **<Interfacename>__Supply__** (nicht-operationaler Anteil) sowie Klassenmethoden für die Operationen des Interfacetyps im Entwurfsmodell selbst (operationaler Anteil).
Die lokale Klasse **__User** beinhaltet in Analogie alle diejenigen Methoden, die für Operationen der CORBA-IDL-Interfacedefinitionen des CORBA-IDL-Moduls **<Interfacename>__Use__** generiert wurden (ausschließlich nicht-operationaler Anteil).
Alle Klassenmethoden besitzen die gleiche Signatur wie die entsprechenden *Servant*-Klassenmethoden (vgl. *Regel 38*), ergänzt um einen zusätzlichen Parameter vom Typ **const PortableServer::ServantBase***.

Die Methoden besitzen jeweils einen zusätzlichen Parameter des Typs **const PortableServer::ServantBase***, um die Instanz der rufenden *Servant*-Klasse zur Ausführungszeit identifizieren zu können. Diese Identifikation wird zur dynamischen Bestimmung der an einer Interaktion beteiligten *Port*-Definition genutzt.

Die Interface-*Composition*-Klasse erhält einen Konstruktor. In der Implementierung dieses Konstruktors werden lokale *Member*-Variablen initialisiert, die Instanzen der lokalen Klassen **__Supply** bzw. **__Use** aufnehmen können. Weiterhin werden Methoden definiert, die den Zugriff auf diese *Member*-Variablen ermöglichen. Deren Implementierung ist so gestaltet, daß beim ersten Ruf dieser Methoden eine **__Supply**- bzw. **__Use**-Klasseninstanz erzeugt und in den jeweiligen lokalen *Member*-Variablen hinterlegt wird, die im folgenden durch diese Klassenmethoden zurückgegeben wird.

Damit die Instanzen der lokalen Klassen die Instanz der Interface-*Composition*-Klasse identifizieren können, von der sie erzeugt wurden, erhalten auch die lokalen Klassen Konstruktoren, in denen Zeiger auf die Interface-*Composition*-Klasse übergeben werden. Zur Hinterlegung dieser Zeiger und zum Zugriff werden wiederum geeignete Klassenvariablen und -methoden an den lokalen Klassen bereitgestellt.

(Beispiel 32) Für die in *Beispiel 28* dargestellte Situation wird für ***i*** die folgende Interface-*Composition*-Klasse erzeugt:

```
namespace M1 {
  class iComposition {
  public :
    class __Supplier {
      const iComposition * _composition;
    public :
      __Supplier ( const iComposition * composition );
```

```
      void op ( const PortableServer::ServantBase * ) const;
      void push_receiveSig
        ( ::M1::Sig * signal, const PortableServer::ServantBase * ) const;
    };
    class __User {
      const iComposition * _composition;
    public :
      __User ( const iComposition * composition );
      void push_sendSig
        ( ::M1::Sig * signal, const PortableServer::ServantBase * ) const;
    };
  private :
    __Supplier * __supplier;
    __User * __user;
  public :
    iComposition();
    virtual ~iComposition();
    const __Supplier * supplier ();
    const __User * user ();
  };
}
```

Zusätzlich wird mit der folgenden Regel die Delegierung der **acquire**-Operationen für CORBA-IDL-Interfacedefinitionen spezifiziert, die für *Provided-Port*-Definitionen mit der Auszeichnung *Multiple* im Entwurfsmodell erzeugt wurden.

(Regel 43) Falls die zur Interface-*Composition*-Klasse korrespondierende Instanz des Konzeptes Interfacetyp im Entwurfsmodell im Kontext einer *Provided-Port*-Definition mit der Auszeichnung *Multiple* auftritt, wird für jede Interface-*Composition*-Klasse eine Methode **`acquire`** definiert, die das Ziel der Delegierung von **`acquire`**-Operationsrufen an *Servant*-Instanzen ist, die die entsprechend *Regel 35* erzeugte CORBA-IDL-Interfacedefinition **<Interfacename>__Dynamic** implementieren.

Im folgenden können nun die Ableitungsregeln für CO-Typ-*Composition*-Klassen formuliert werden. Diese Klassen stellen den Kontext eines CO-Typs in der programmiersprachlichen Realisierung in $CORE_{WARE}$ her. Die strukturellen Zusammenhänge zwischen *Servant*-Klassen sowie Interface- und CO-Typ-*Composition*-Klassen sind in Abb. 21 dargestellt.

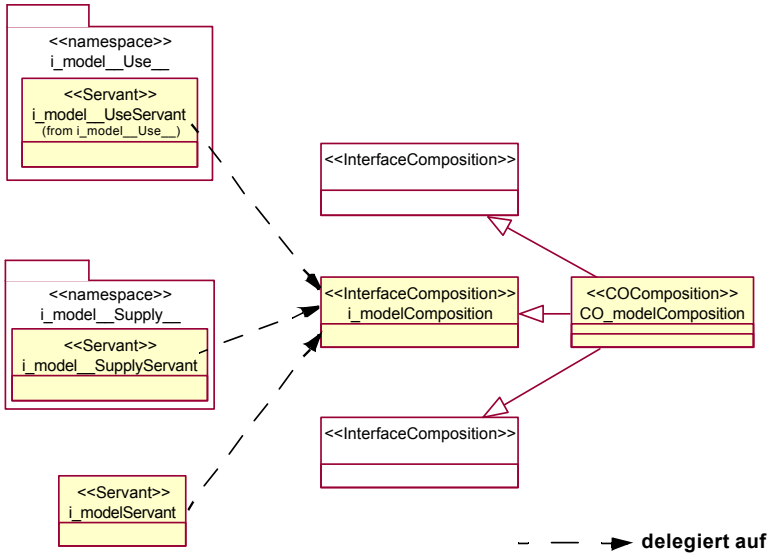


Abb. 21 Delegations- und Vererbungsrelationen zwischen erzeugten C++-Klassen

(Regel 44) Für jede Instanz des Konzeptes CO-Typ (CO-Typ im Entwurfsmodell) wird eine korrespondierende C++-Klasse mit dem Namen **<CO-Typname>Composition** in einem C++-Namensraum erzeugt. Dieser Namensraum korrespondiert zu dem Namensraum im Entwurfsmodell, in dem der CO-Typ definiert ist. Diese C++-Klasse spezialisiert diejenigen Interface-*Composition*-Klassen, die für Interfacetypen im Entwurfsmodell erzeugt wurden, zu denen der CO-Typ mindestens eine *Provided*- oder eine *Used-Port*-Definition besitzt.

Damit ist sichergestellt, daß Instanzen von *Servant*-Klassen, die für Interfacetypen im Entwurfsmodell erzeugt wurden, Operationsrufe an diese, das CO selbst repräsentierende CO-Typ-*Composition*-Klasseninstanz delegieren. Insbesondere werden im Falle von *Provided-Port*-Definitionen mit der Auszeichnung *Multiple* im Entwurfsmodell die entsprechenden **aquire**-Operationsrufe ebenfalls an diese CO-Typ-*Composition*-Klasse weitergeleitet.

2.3.1.3 Repräsentation von Provided-Port-Definitionen in CO-Typ-Composition-Klassen

Servant-Klassen, die die einen CO-Typ repräsentierende CORBA-IDL-Interfacedefinition implementieren, delegieren Aufrufe der für sie definierten Methoden (Interfacenavigation, Reflexion, etc.) an die CO-Typ-*Composition*-Klasse. Diese Klasse ist für die Instanziierung von *Servant*-Klasseninstanzen verantwortlich. Die Instanziierung basiert auf den *Port*-Definitionen für den CO-Typ, den diese *Composition*-Klasse repräsentiert.

(Regel 45) Für jede Instanz des Konzeptes *Provided-Port*-Definition ohne die Auszeichnung *Multiple* werden in Abhängigkeit von den Interaktionselementen des zugehörigen Interfacetyps im Entwurfsmodell *Servant*-Instanzen erzeugt.
Es wird eine Instanz derjenigen *Servant*-Klasse erzeugt, die für die CORBA-IDL-Interfacedefinition produziert wurde, die den Interfacetyp im Entwurfsmodell repräsentiert (operationaler Interfaceanteil).
Falls zusätzlich Signal- oder *Continuous-Media*-Interaktionselemente für den Interfacetyp im Entwurfsmodell definiert sind, wird weiterhin eine Instanz derjenigen *Servant*-Klasse angelegt, die für die CORBA-IDL-Interfacedefinition mit dem Namen **<Interfacename im Entwurfsmodell>__Supply** entsprechend *Regel 27* erzeugt wurde. Die Instanziierung erfolgt im Konstruktor der CO-Typ-*Composition*-Klasse, Zeiger auf die erzeugten Instanzen werden in *Member*-Variablen dieser Klasse mit den Namen

- **_<Name der Provided-Port-Definition>** bzw.
- **_<Name der Provided-Port-Definition>_supply**

hinterlegt. Zusätzlich wird eine Instanz der *Servant*-Klasse für den CO-Typ selbst erzeugt und in der *Member*-Variablen mit dem Namen **_<CO-Typ>** abgelegt.
Die erzeugten Instanzen werden durch den Destruktor der CO-Typ-*Composition*-Klasse entfernt.

(Beispiel 33) Für die in *Beispiel 28* dargestellte Situation ergeben sich für die *Provided-Port*-Definition mit dem Namen ***serve*** die folgenden C++-Definitionen für die CO-*Composition*-Klasse:

```
namespace M1 {
 class OComposition
  :  public ::M1::iComposition {
 private :
  OServant * _O;
  ::M1::iServant * _serve;
  ::M1::i__Supply__::i__SupplyServant * _serve_supply;
 public :
  OComposition();
  virtual ~OComposition();
  ::M1::i_ptr provide_serve ( const PortableServer::ServantBase * ) const;
  ::M1::i__Supply__::i__Supply_ptr provide_serve__Supply
     ( ::ComponentModel::Parameters& params,
       const PortableServer::ServantBase * ) const;
  };
}
```

2.3.1.4 Repräsentation von Used-Port-Definitionen in CO-Typ-Composition-Klassen

In Analogie zu *Provided-Port*-Definitionen verwaltet die CO-Typ-*Composition*-Klasse auch die CORBA-Interfaces und die instanziierten *Servant*-Klassen für *Used-Port*-Definitionen. Dabei werden - den Ableitungsregeln der Konfigurationssicht (Abschnitt 2.2) folgend - Implementierungen für **connect**-Operationen definiert. Während des Rufes einer **connect**-Operation wird die übergebene CORBA-Interfacereferenz für das den operationalen Anteil eines Interfacetyps repräsentierende CORBA-IDL-Interface in einer *Member*-Variablen hinterlegt. Darüber hinaus wird im Falle von zusätzlicher Signal- bzw. *Continuous-Media*-Interaktion eine *Servant*-Klasseninstanz für den *Requires*-Fall instanziiert und ebenfalls in einer *Member*-Variablen hinterlegt. Diese Darstellung wird durch die folgenden Regel präzisiert:

(Regel 46) Für eine Instanz des Konzeptes *Used-Port*-Definition ohne Auszeichnung *multiple* eines CO-Typs wird in der CO-Typ-*Composition*-Klasse eine Methode **connect_<Port-Name>** definiert. Die Signatur dieser Methode ist durch die Anwendung von [OMG C++Map] auf die durch *Regel 29* und *Regel 32* resultierenden CORBA-IDL-Definitionen gegeben, erweitert um einen Parameter vom Typ **::PortableServer::ServantBase *.** Neben der Klassenmethode definiert die CO-Typ-*Composition*-Klasse

- eine *Private-Member*-Variable vom Typ der CORBA-IDL-Interfacedefinition für den operationalen Anteil des Interfacetyps der *Used-Port*-Definition,
- Zugriffsmethoden zum Setzen und Lesen dieser *Member*-Variablen sowie
- eine *Private-Member*-Variable vom Typ der *Servant*-Klasse für den nicht-operationalen Anteil des Interfacetyps.

Die Abläufe während des Rufes einer **connect**-Operation für eine *Used-Port*-Definition werden sukzessive präzisiert. Im folgenden werden - analog zur Darstellung der Ableitungsregeln für *Provided-Port*-Definitionen - die Klassendefinitionen der CO-Typ-*Composition*-Klasse anhand eines Beispiels präsentiert.

(Beispiel 34) Die Klassendeklaration für die CO-Typ-*Composition*-Klasse aus *Beispiel 33* wird entsprechend *Regel 46* um Deklarationen für die *Used-Port*-Definition mit dem Namen ***use*** ergänzt:

```
namespace M1 {
 class OComposition
  :  public ::M1::iComposition {
  ::M1::i_var _use;
  ::M1::i__Use__::i__UseServant * _use_use;
 public :
  ::M1::i_ptr get_use () const;
  void set_use ( ::M1::i_ptr );
 public :
  ::M1::i__Use__::i__Use_ptr connect_use
    (  ::M1::i_ptr connect_use_p,
       ::ComponentModel::Parameters& params,
       const PortableServer::ServantBase * ) const;
 };
}
```

2.3.1.5 Repräsentation von Multiple-Port-Definitionen

Es lassen sich nun die Ableitungsregeln für *Multiple-Port*-Definitionen formulieren. Die **connect**-Operation der CORBA-IDL-Interfacedefinition, die den CO-Typ repräsentiert, wurde gerade so definiert, daß sowohl *Single*- als auch *Multiple-Port*-Definitionen mittels der gleichen Signatur behandelt werden können (vgl. Abschnitt 2.2.3.2). Dementsprechend kann die Klassenmethode der CO-Typ-*Composition*-Klasse, die diese **connect**-Operation repräsentiert, ebenfalls beide Fälle behandeln. Es bleibt also zu untersuchen, auf welche Weise *Provided-Port*-Definitionen in Entwurfsmodellen durch $CORE_{MAP}$ behandelt werden, die die Auszeichnung *Multiple* tragen.

(Regel 47) Für eine Instanz des Konzeptes *Provided-Port*-Definition mit der Auszeichnung *Multiple* wird im Kontext der CO-Typ-*Composition*-Klasse eine Instanz der *Servant*-Klasse **<Interfacename>__DynamicServant** angelegt und in einer *Member*-Variable mit dem Namen **_<Multiple-Port-Name>** hinterlegt. Als Ziel der Delegierung der **acquire**-Operationsimplementierung dieser *Servant*-Klasse wird in der Klasse **<CO-Typ>Composition** eine lokale Klasse **<Interfacename>__DynamicDelegation** erzeugt und während der Konstruktion von **<CO-Typ>Composition** angelegt. Auf diese kann mittels der Methode **<Interfacename>__Dynamic()** von **<CO-Typ>Composition** zugegriffen werden. Die lokale Klasse definiert einen Konstruktor, der eine Instanz der Klasse **<CO-Typ>Composition** übergeben bekommt. Weiterhin deklariert sie die Methode **acquire**, deren Signatur sich durch Anwendung von [OMG C++ Map] auf die CORBA-IDL-Interfacedefinition **<Interfacename>__Dynamic** im CORBA-IDL-Modul **__Supply__** ergibt und um einen Parameter **::PortableServer::ServantBase *** erweitert wird.

Abb.22 illustriert die Abbildungs-, Delegierungs- und Vererbungsrelationen für *Provided-Port*-Definitionen mit der Auszeichnung *Multiple*. Dabei entsteht durch Anwendung von *Regel 34* für solche Definitionen für einen Interfacetyp ***i_model*** die CORBA-IDL-Interfacedefinition **i_model__Dynamic** im CORBA-IDL-Modul **__Supply__**. Diese Interfacedefinition wird durch die entsprechend *Regel 39* produzierte *Servant*-Klasse **i_model__DynamicServant** implementiert, die zur Realisierung der Implementierung das Entwurfsmuster Delegierung an die Interface-*Composition*-Klasse **i_modelComposition** benutzt.

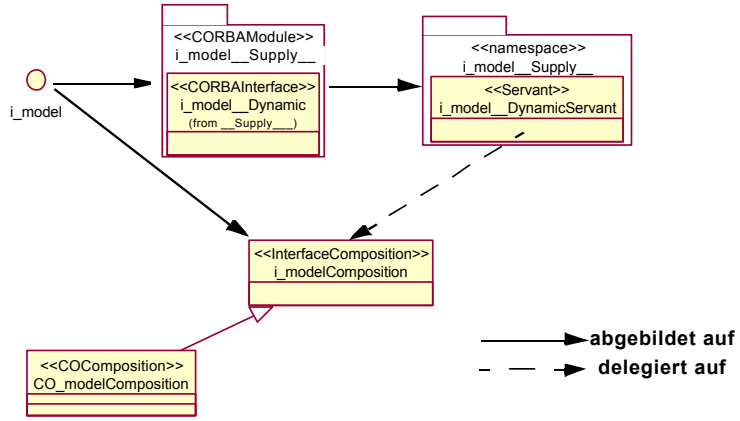


Abb. 22 Abbildungs-, Delegierungs- und Vererbungsrelationen im Falle einer Multiple-Port-Definition

Die Abbildung von *Provided-Port*-Definitionen mit der Auszeichnung *Multiple* folgt offensichtlich nicht dem allgemeinen Abbildungsschema, daß *Servant*-Klassen für CORBA-IDL-Operationen, die im Kontext eines CO-Typs definiert sind, an die **<CO-Typ>Composition**-Klasse delegieren. Die Anwendung von *Regel 47* impliziert die Erzeugung von lokalen Klassen für jeden Interfacetyp, auf den solche *Provided-Port*-Definitionen verweisen. Der Grund ist die Signatur der **acquire**-Operation, die sich nur dem Rückgabetyp nach zwischen verschiedenen Interfacetypen unterscheidet, dementsprechend würden mehrere **acquire**-Klassenmethoden ohne Veränderung der Signatur dieser Methoden inkorrekte C++-Deklarationen ergeben[1].

1. In C++ ist das Konzept *Co-variant Pointer* aufgenommen worden, das die Definition solcher, nur dem Rückgabetyp nach verschiedener Klassenmethoden erlaubt. Dies wird jedoch nicht von allen *Compiler*-Werkzeugen unterstützt.

(Beispiel 35) Um die Wirkungsweise von *Regel 47* darzustellen, wird das *Beispiel 28* insofern verändert, als das die *Provided-Port*-Definition **serve** nun die Auszeichnung *Multiple* erhält. Dann werden die folgenden zusätzlich erzeugten bzw. zu *Beispiel 28* veränderten C++-Klassen produziert:

```
namespace M1 {
 namespace i__Supply__ {
  class i__DynamicServant : public virtual POA_M1::i__Supply__::i__Dynamic {
   const ::M1::OComposition * _delegator;
  public:
   i__DynamicServant ( const ::M1::OComposition * delegator );
   ::M1::OComposition * get_delegator () const;
   virtual ::M1::i__Supply__::i_Access * acquire
      ( const ::ComponentModel::Parameters& params );
  };
 }

 class OComposition : public ::M1::iComposition {
 public :
  class i__DynamicDelegation {
   const OComposition * _delegator;
  public :
   i__DynamicDelegation ( const OComposition * delegator );
   ::M1::i__Supply__::i_Access * acquire
      ( const ::ComponentModel::Parameters& params,
        const PortableServer::ServantBase * ) const;
  };
 private :
  OServant * _O;
  const i__DynamicDelegation * _i__Dynamic;
  ::M1::i__Supply__::i__DynamicServant * _serve;
 public :
  OComposition();
  virtual ~OComposition();
  ::M1::i__Supply__::i__Dynamic_ptr provide_serve
     ( const PortableServer::ServantBase * ) const;
  const i__DynamicDelegation * i__Dynamic () const;
 };
}
```

Ein wesentlicher Verhaltensaspekt der CO-Typrealisierung ist die Implementierung der Interfacenavigation, d.h. letztlich in der Verfügbarkeit des *Port*-Management für externe Klienten. Die externen Interaktionspunkte für die Interfacenavigation sind bereits durch die CORBA-IDL-Repräsentation eines CO-Typs vorgegeben, für die entsprechend *Regel 40 Servant*-Klassen produziert und während der Konstruktion der Klasse **<CO-Typ>Composition** Instanzen erzeugt wurden (vgl. *Regel 44* und *Regel 45*). In *CORE* soll die Implementierung der Interfacenavigation durch die produzierte CO-Typ-*Composition*-Klasse selbst realisiert werden, d.h. ohne Beteiligung der durch den Entwickler bereitgestellten Implementierung der Artefaktrepräsentationen. Es ist also zu untersuchen, auf welche Weise diese die externen Interaktionspunkte realisierenden *Servant*-Klassen implementiert werden.

(Regel 48) Die *Servant*-Klasse, die für die den CO-Typ repräsentierende CORBA-IDL-Interfacedefinition erzeugt (*Regel 40*) und entsprechend *Regel 44* und *Regel 45* durch den Konstruktor der CO-Typ-*Composition*-Klasse instanziiert wird, delegiert alle eintreffenden Interfacenavigationsrufe der Art **provide_<Port-Name>**, **provide_<Port-Name>__Supply** bzw. **connect_<Port_Name>** an die CO-Typ-*Composition*-Klasse.

Da die CO-Typ-*Composition*-Klasse die *Servant*-Instanzen für alle Repräsentationen von *Used*- oder *Provided-Port*-Definitionen selbst verwaltet, implementiert sie die Interfacenavigationsrufe durch Rückgabe einer CORBA-Interfacereferenz der entsprechenden *Servant*-Instanz. Darüber hinaus werden während der Abarbeitung der **provide**- bzw. **connect**-Methoden der CO-Typ-*Composition*-Klasse die Kommunikationskanäle für Signal- bzw. *Continuous-Media*-Interaktionselemente durch $CORE_{WARE}$ konfiguriert. Die konkreten Mechanismen sind Bestandteil des Interaktionsmanagement und werden im Zusammenhang mit der durch

CORE$_{MAP}$ hergestellten Korrelation zwischen Artefaktrepräsentationen und Interaktionselementen untersucht.

(Beispiel 36) Für *Beispiel 28* werden für die mit ***serve*** bezeichnete *Provided-Port*-Definition im Entwurfsmodell folgende Implementierungen für **provide**-Klassenmethoden der *Servant*-Klassen produziert:

```
namespace M1 {
  ::M1::i_ptr OServant::provide_serve () {
    return this -> get_delegator() -> provide_serve (  this );
  }

  ::M1::i__Supply__::i__Supply_ptr OServant::provide_serve__Supply
     ( ::ComponentModel::Parameters& params ) {
    return this -> get_delegator() -> provide_serve__Supply ( params, this );
  }
}
```

(Beispiel 37) Analog zu *Beispiel 36* erhält man durch Anwendung von *Regel 48* auf die mit ***use*** bezeichnete *Used-Port*-Definition die folgende Implementierung der *Servant*-Klasse für **OServant**:

```
module M1 {
  ::M1::i__Use__::i__Use_ptr OServant::connect_use
     ( ::M1::i_ptr connect_use_p, ::ComponentModel::Parameters& params ) {
    return this -> get_delegator()
       -> connect_use ( connect_use_p, params, this );
  }
}
```

(Regel 49) Die Implementierung der Interfacenavigationsmethoden für *Provided-* und *Used-Port*-Definitionen mit der Auszeichnung *Multiple* erfolgt analog zu *Regel 48*.

Zur Realisierung des Delegierungskonzepts erhalten *Servant*-Klassen in ihren Konstruktoren C++-Zeiger auf die korrespondierenden *Composition*-C++-Klassen als Parameter:

- *Servant*-Klassen für CORBA-IDL-Interfacedefinitionen, die für Interfacetypen im Entwurfsmodell produziert werden, erhalten C++-Zeiger auf Interface-*Composition*-Klassen,
- *Servant*-Klassen für CORBA-IDL-Interfacedefinitionen, die für CO-Typen im Entwurfsmodell produziert werden, erhalten C++-Zeiger auf CO-Typ-*Composition*-Klassen.

Die übergebenen Zeiger werden in einer lokalen *Member*-Variablen der *Servant*-Klassen hinterlegt und können mittels einer ebenfalls durch *CORE*$_{MAP}$ erzeugten Methode **get_delegator** gelesen werden.

Die Benutzung der Interfacenavigationsmethoden setzt für Klienten jedoch die Kenntnis aller CORBA-IDL-Interfacedefinitionen voraus, die die angebotenen Interfacetypen im Entwurfsmodell repräsentieren. Um einen allgemeinen Mechanismus anzubieten, der dem Klienten den Zugriff über eine *Provided-Port*-Definition auf ein spezifisches CORBA-Interface erlaubt, wird die CORBA-IDL-Interfacedefinition **ComponentModel::ComponentBase** um generische Operationen **provide**, **provide_supply** und **connect** erweitert:

```
module ComponentModel {
  interface ComponentBase {
    // ...
    Object provide ( in string port );
    Object provide_supply ( in string port, inout Parameters params );
    Object connect ( in string port, in Object provided, inout Parameters params );
  };
};
```

Auch bei der Definition dieser zusätzlichen Operationen wurde das in den Abschnitten 2.2.1 und 2.2.2 diskutierte Abbildungsschema angewandt. Dementsprechend werden hier ebenfalls zwei Operationen **provide** und **provide_supply** für *Provided-Port*-Definitionen im Entwurfsmodell und eine Operation **connect** für *Used-Port*-Definitionen im Entwurfsmodell definiert. Den zusätzlichen Operationen von **ComponentBase** wird mittels des Parameters **port** der Name der *Port*-Definition der *Used-* bzw. *Provided-Port*-Definition übergeben, der Parameter **params** dient wiederum der Konfiguration der Signal- und *Continuous-Media*-Kommunikationska-

näle durch *CORE$_{WARE}$*. Der Parameter **provided** enthält die Referenz des CORBA-Interfaces (operationaler Anteil), das im Kontext einer *Used-Port*-Definition hinterlegt wird.

Die Deklaration der *Servant*-Klassen für CO-Typen verändert sich damit kanonisch entsprechend *Regel 40*, so daß sich für die Situation in *Beispiel 28* die folgenden Klassenmethodendeklarationen für die **OServant**-Klasse ergeben:

```
namespace M1 {
 class OServant : public virtual POA_M1::O {
  virtual CORBA::Object_ptr provide ( const char * );
  virtual CORBA::Object_ptr provide_supply
    ( const char *, ::ComponentModel::Parameters& params );
  virtual CORBA::Object_ptr connect
    ( const char * port, CORBA::Object_ptr provided,
      ::ComponentModel::Parameters& params );
 };
}
```

Die Implementierung der konkreten Interfacenavigationsmethoden (Interfacenavigation mit durch *CORE$_{MAP}$* produzierten CORBA-Interfacedefinitionen) wird durch die CO-Typ-*Composition*-Klasse vorgenommen. Im Gegensatz dazu kann die Implementierung der generischen Interfacenavigationsoperationen bereits in der *Servant*-Klasse realisiert werden, die für einen CO-Typ selbst entsprechend *Regel 40* produziert wird.

(Regel 50) Die Operationen **provide**, **provide_supply** und **connect** werden durch die dem CO-Typ entsprechende *Servant*-Klasse implementiert (vgl. *Regel 40*). Diese Implementierung vergleicht den als Parameter **port** übergebenen Namen mit allen an dem CO-Typ im Entwurfsmodell spezifizierten Namen von *Provided-* bzw. *Used-Port*-Definitionen. Im Falle von Gleichheit wird die in der gleichen Klasse definierte zugehörige Methode **provide_<Port-Name>**, **provide_supply_<Port-Name>** bzw. **connect_<Port-Name>** gerufen.

(Beispiel 38) Die Implementierungen von **provide**, **provide_supply** und **connect** in der Klasse **OServant** für die in *Beispiel 28* beschriebene Situation sind entsprechend *Regel 50* gegeben durch:

```
namespace M1 {
 CORBA::Object_ptr OServant::provide ( const char * port ) {
  if ( strcmp ( port, "serve" ) == 0 )
    return provide_serve();
  return CORBA::Object::_nil();
 }

 CORBA::Object_ptr OServant::provide_supply
  (  const char * port,
     ::ComponentModel::Parameters& params ) {
  if ( strcmp ( port, "serve" ) == 0 )
    return provide_serve__Supply ( params );
  return CORBA::Object::_nil();
 }

 CORBA::Object_ptr OServant::connect
  (  const char * port,
     CORBA::Object_ptr provided,
     ::ComponentModel::Parameters& params ) {
  if ( strcmp ( port, "use" ) == 0 )
  {
    return connect_use ( ::M1::i::_narrow ( provided ), params );
  }
  return CORBA::Object::_nil();
 }
}
```

Prinzipiell werden die gleichen Mechanismen angewendet, um Interaktionselemente, die nicht der Interfacenavigation zuzuordnen sind, per Delegierung innerhalb der *Servant*-Klassenimplementierung umzusetzen. Diese Interaktionselemente werden durch Interface-*Composition*-Klassen realisiert. Die entsprechenden

Servant-Klassen haben - definiert durch *Regel 38* bzw. *Regel 39* - ebenfalls Zugriff auf die in Korrelation zu ihnen stehenden Interface-*Composition*-Klassen. Der Unterschied zur Delegierung im Falle der Interfacenavigation besteht in der Unterscheidung, ob eine *Servant*-Klasse den *Supply*- (*Supports*-Relation zwischen CO-Typ und Interfacetyp) oder *Use*-Fall (*Requires*-Relation zwischen CO-Typ und Interfacetyp) implementiert.

(Regel 51) Die Implementierung der entsprechend *Regel 38* bzw. *Regel 39* erzeugten *Servant*-Klassenmethoden für Interfacetypen im Entwurfsmodell erfolgt in Abhängigkeit vom *Supply*- bzw. *Use*-Fall durch Delegierung an die lokalen Klassen **__Supplier** bzw. **__User** der zu dem Interfacetyp korrelierten Interface-*Composition*-Klasse. Die Operationen der CORBA-IDL-Interfacedefinition **::ComponentModel::ConsumerBase** werden durch die entsprechende *Servant*-Klasse durch Auswertung des empfangenen Signals und Delegierung an sich selbst implementiert.

(Beispiel 39) Exemplarisch sei hier die Implementierung der *Servant*-Klassen für die Interaktionselemente ***op*** für den *Supply*-Fall und ***sendSig*** für den *Use*-Fall entsprechend *Beispiel 28* dargestellt[1]:

```
namespace M1 {
 namespace i__Use__ {
  void sendSig_consumerServant::push_sendSig ( ::M1::Sig * signal ) {
    this -> get_delegator() -> user() -> push_sendSig ( signal, this );
  }
 }
 void iServant::op () {
  this -> get_delegator() -> supplier() -> op (  this );
 }
}
```

Die erzeugte Implementierung für die Operationen der Basisinterfacedefinition **::ComponentModel::ConsumerBase** wurde aus Gründen der besseren Performanz des generierten Codes nicht durch Delegierung an die Interface-*Composition*-Klasse, sondern durch Typauswertung des empfangenen Signals und Delegierung an sich selbst vorgenommen.

(Beispiel 40) Für den Interfacetyp ***i***, werden die **push**-Operation im *Use*- bzw. *Supply*-Fall entsprechend *Regel 51* durch die *Servant*-Klassen für ***i*** folgendermaßen implementiert:

```
namespace M1 {
 namespace i__Use__ {
  void sendSig_consumerServant::push ( const CORBA::Any& any) {
    ::M1::Sig * event_;
    if ( any >>= event_ )
     this->push_sendSig ( event_ );
  }
 }

 namespace i__Supply__ {
  void receiveSig_consumerServant::push ( const CORBA::Any& any) {
    ::M1::Sig * event_;
    if ( any >>= event_ )
     this->push_receiveSig ( event_ );
  }
 }
}
```

Die bisher vorgestellten Ableitungsregeln für Konzepte der Implementierungssicht realisieren die Kopplung einer CO-Typ-Realisierung an $CORE_{WARE}$. Im einzelnen können mit diesen Ableitungsregeln *Servant*-Klassen produziert werden, die Operationsrufe für alle CORBA-IDL-Interfacedefinitionen der externen Sicht entgegennehmen und diese an entsprechende Interface-*Composition*-Klassen delegieren. Diese Interface-*Composition*-Klassen stellen, wie eingangs gefordert, den gemeinsamen Kontext aller an einem Interfacetyp definierten Interaktionselemente in der Implementierung her. Die für einen CO-Typ selbst erzeugte

1. Nutzung des Interfacetyps ***i*** heißt dabei für das Interaktionselement ***sendSig***, definiert als *Produce*-Definition für ***Sig***, die Realisierung des Empfangs von Signalen des Typs ***Sig***.

Composition-Klasse stellt wiederum den Zusammenhang zwischen CO-Typdefinition und Interfacetypen im Entwurfsmodell her, zwischen denen eine *Supports*- bzw. *Requires*-Relation besteht. Darüber hinaus bindet diese *Composition*-Klasse die Implementierung der Interfacenavigation an $CORE_{WARE}$ an. Damit sind die Voraussetzungen geschaffen, die das Verhalten eines CO-Typs erbringenden Artefakte programmiersprachlich zu repräsentieren und an die CO-Typ-Repräsentation anzubinden.

2.3.1.6 Spezifika der Abbildung von Produce- und Consume-Interaktionselementen

Die bisher diskutierten Ableitungsregeln basieren auf den CORBA-IDL-Interfacedefinitionen, die entsprechend der Abbildung der Konzepte der Struktur- und Konfigurationssicht durch $CORE_{MAP}$ produziert werden. Der Darstellung der Abbildung von *Produce*-Interaktionselementen im *Supports*-Fall bzw. *Consume*-Interaktionselementen im *Requires*-Fall kann entnommen werden, daß für diese Interaktionselemente lokale Interfacedefinitionen produziert werden, die durch die bisher vorgestellten Ableitungsregeln nicht erfaßt wurden. Die programmiersprachliche Repräsentation von Artefakten ist aber auf die Verfügbarkeit von Mechanismen in $CORE_{WARE}$ angewiesen, die das Versenden von Signalen gestatten. Dazu ist es notwendig, die CORBA-IDL-Interfacedefinitionen, die entsprechend *Regel 11* produziert werden, in programmiersprachliche Konstrukte zu überführen. Diese Konstrukte sind **`SignalProducer`**-Klassen, die durch $CORE_{MAP}$ erzeugt werden.

(Regel 52) Für jede mittels *Regel 11* produzierte lokale CORBA-IDL-Interfacedefinition wird eine Klasse mit dem Namen **`<Name des lokalen CORBA-IDL-Interfaces>Producer`** erzeugt. Diese Klasse spezialisiert die durch $CORE_{WARE}$ definierte Klasse **`Container::SignalProducer`**. Sie definiert die Klassenmethode **`push_<produce- bzw. consume-Name>`**. Die Implementierung dieser Methode erfolgt durch Nutzung des CORBA-IDL-Datentyps **`any`** und der **`push`**-Methode der Basisklasse **`SignalProducer`**. Die Klasse **`<Name des lokalen CORBA-IDL-Interfaces>Producer`** definiert einen Konstruktor, der die Parameter vom Typ **`const ComponentModel::ComponentBaseImpl *`** und **`CosEventChannelAdmin::EventChannel_ptr`** an den Konstruktor der Basisklasse übergibt.

Die Klasse **`SignalProducer`** von $CORE_{WARE}$ definiert allgemeine Mechanismen der Konfiguration von CORBA-*Event*- bzw. *Notification*-Kanälen sowie generische Methoden zum Versenden von Signalen. Insbesondere stellen deren Konstruktoren die technologiespezifische Realisierung des Anschlusses einer **`SignalProducer`**-Klasseninstanz an einen CORBA-*Event*- bzw. *Notification*-Kanal bereit. Darüber hinaus definiert diese Klasse die Methode **`void push(const CORBA::Any& data)`**, die den plattformspezifischen Mechanismus des Versendens eines Signals via zuvor angeschlossenem *Event*- bzw. *Notification*-Kanal realisiert, so daß die Spezialisierungen von **`SignalProducer`** diese Methode zum Versenden konkreter Signale in Form des CORBA-IDL-Datentyps **`any`** nutzen können.

(Beispiel 41) Exemplarisch sei für *Beispiel 28* die Erzeugung der Klasse **`sendSig_Producer`** dargestellt, die das Versenden von Signalen des Signaltyps ***`Sig`*** im Kontext des *Produce*-Interaktionselements ***`sendSig`*** auf der den Interfacetyp ***`i`*** bereitstellenden Seite realisiert. Die Deklaration der Klasse **`M1::i__Use__::receiveSig_Producer`** erfolgt analog.

```
namespace M1 {
 namespace i__Supply__ {
  class sendSig_Producer : public Container::SignalProducer {
  public :
    sendSig_Producer (
     const ComponentModel::ComponentBaseImpl *,
     CosEventChannelAdmin::EventChannel_ptr );
    virtual ~sendSig_Producer();
    virtual void push_sendSig ( ::M1::Sig * signal );
  };
 }
}
```

Die Implementierung der Klassenmethode **push_sendSig** nutzt die Methode **push** der durch *CORE*$_{WARE}$ bereitgestellten Basisklasse **SignalProducer**, die mittels der Konstruktionsparameter den Anschluß an einen vorgegebenen *Event-* bzw. *Notification*-Kanal herstellt.

(Beispiel 42) Exemplarisch sei hier die Implementierung der Klasse **sendSig_Producer** aus *Beispiel 41* dargestellt.

```
namespace M1 {
 namespace i__Supply__ {
  sendSig_Producer::sendSig_Producer
    ( const ComponentModel::ComponentBaseImpl * co,
     CosEventChannelAdmin::EventChannel_ptr channel )
     : Container::SignalProducer ( co, channel )
  {}

  void sendSig_Producer::push_sendSig ( ::M1::Sig * signal ) {
    CORBA::Any any;
    any <<= signal;
    this -> push ( any );
  }
 }
```

Die resultierenden C++-Klasseninstanzen für die Signalinteraktion und ihre Anbindung an einen CORBA-*Event-* oder *Notification*-Kanal unter Nutzung der Definition des *Produce*-Interaktionselements ***sendSig*** aus *Beispiel 28* illustriert Abb.23. Das Signal ***sendSig*** wird von einem Anbieter des Interfacetyps ***i*** produziert und von einem Nutzer dieses Interfacetyps konsumiert. In diesem Szenario nutzt die Realisierung des Implementierungselements für ***sendSig*** im Kontext der Artefaktrepräsentation die Methode **push_sendSig** einer Instanz der entsprechend *Regel 52* erzeugten Klasse **sendSig_Producer** zum typsicheren (Signalparameter vom Typ **::M1::Sig ***) Verschicken eines ***Sig***-Signals. Diese Instanz wiederum übergibt das Signal in Form der CORBA-IDL-**any**-Repräsentation an die Methode **push** der Basisklasse

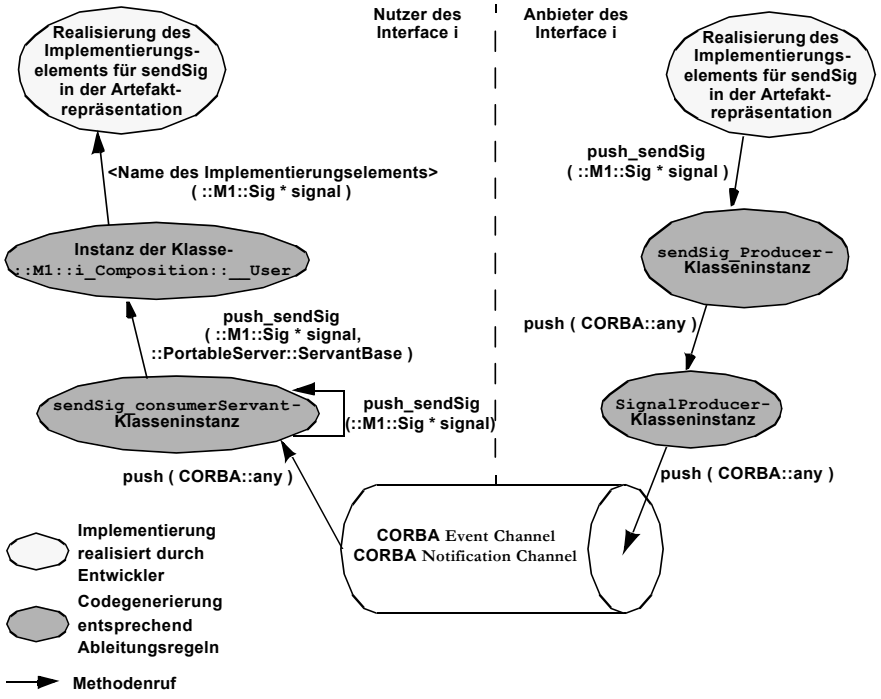


Abb. 23 Kooperation von Servant-, Composition- und Signal-Producer-Klasseninstanzen bei der Realisierung der Signalinteraktion

::Container::SignalProducer, die das Signal via konfiguriertem CORBA-*Event*- bzw. CORBA-*Notification*-Kanal weitertransportiert. Auf der empfangenden Seite wird die Methode **push** an einer Instanz der Klasse **sendSig_consumerServant** gerufen, deren Realisierung aus der CORBA-IDL-**any**-Repräsentation eine Instanz des Typs **::M1::Sig** erzeugt und diese an die entsprechende Instanz der Interface-*Composition*-Klasse übergibt. Die Implementierung dieser Klasse nutzt das Interaktions- und Artefaktmanagement, um das Signal an ein Implementierungselement einer Artefaktrepräsentation auszuliefern. Abschließend sei hervorgehoben, daß aufgrund der Interfacedefinitionen der *Continuous-Media-Delivery*-Plattform (vgl. [KKS 00]) im Gegensatz zur Signalinteraktion keine spezifischen *Servant*- bzw. *Composition*-Klassen für *Continuous-Media*-Interaktionselemente zu erzeugen sind.

Mit der Gesamtheit aller bisher vorgestellten Ableitungsregeln für die Konzepte der Implementierungssicht können das *Port*-Management und die Anbindung von CO-Typ-Repräsentationen mit ihren Interaktionspunkten an die *Component-Support*-Plattform realisiert werden. Die Regeln zur Erzeugung der Interface-*Composition*-Klassen stellen dabei den Zusammenhang der *Servant*-Klassen für die verschiedenen Interaktionsarten her. Die Produktion von CO-Typ-*Composition*-Klassen realisiert das *Port*-Management und liefert so den gemeinsamen Kontext der angebotenen und genutzten Interfaces im Rahmen von *Port*-Definitionen.

Im folgenden sollen nun die Mechanismen zur Realisierung der durch $CoRE_{MAP}$ erzeugten dynamischen Korrelation zwischen Implementierungselementen von Artefaktrepräsentationen und Interaktionselementen dargestellt werden. Diese Mechanismen sind integraler Bestandteil des Interaktionsmanagements innerhalb der CO-Typrepräsentation und erbringen diese Funktionalität in Kooperation mit dem Artefaktmanagement, dessen Bestandteile ebenfalls durch $CoRE_{MAP}$ entstehen.

2.3.2 Artefakte und Implementierungselemente

Instanzen des Konzeptes Artefakt in einem Entwurfsmodell repräsentieren die Realisierung des spezifischen Verhaltens von CO-Typen. Diese Spezifik wird durch die Implementierung der Repräsentationen von Implementierungselementen durch den Entwickler bereitgestellt. Die Diskussion der Ableitungsregeln in Abschnitt 2.3.1 resultierte in der Darstellung, auf welche Weise die externen Interaktionspunkte der Repräsentation eines CO-Typs an $CoRE_{WARE}$ gebunden werden, die durch CO-Typen und deren Interfacetypen im Entwurfsmodell definiert wurden. Es wurde präsentiert, wie die mittels Ableitungsregeln erzeugbaren CO-Typ- und Interface-*Composition*-Klassen für Interaktionselemente an Interaktionspunkten Klassenmethoden bereitstellen, die in der Realisierung der *Servant*-Klassen genutzt wurden. Diese Klassenmethoden formen das Interaktionsmanagement von CO-Typrepräsentationen. Darüber hinaus ist das *Port*-Management bereits integraler Bestandteil der durch $CoRE_{MAP}$ produzierten Implementierung von *Servant*- bzw. *Composition*-Klassen. In diesem Abschnitt werden nun die Bestandteile des Artefaktmanagement und der durch $CoRE_{MAP}$ hergestellten Korrelation mit dem Interaktionsmanagement präsentiert. Artefaktdefinitionen in einem Entwurfsmodell spiegeln sich in C++-Klassen wider, deren Klassenmethoden gerade die Implementierungselemente von Artefakten repräsentieren. Dieser allgemeine Abbildungsgedanke wird mittels der folgenden Regel präzisiert:

(Regel 53) Für jede Instanz des Konzeptes Artefakt im Entwurfsmodell wird eine C++-Klasse mit dem Namen des Artefakts in dem C++-Namensraum erzeugt, der dem Namensraum im Entwurfsmodell entspricht, in dem das Artefakt definiert ist. Diese Klasse spezialisiert die abstrakte, durch $CoRE_{WARE}$ definierte Klasse **Container::Artifact**. Die produzierte Klasse erhält einen leeren Konstruktor.

Die Basisklasse **Container::Artifact** deklariert abstrakte, virtuelle Methoden, die zur Identifikation von Artefakten durch das Artefaktmanagement von $CoRE_{WARE}$ genutzt werden. Insbesondere werden diese Methoden für die Anbindung von Instanzen der Repräsentation von Artefakten (Artefaktklasseninstanzen) an die CO-Typ-*Composition*-Klasseninstanzen zur Ausführungszeit genutzt.

Entsprechend den Definitionen des Konzeptraumes in [CoRE II] muß in den Ableitungsregeln für diese C++-Klassen zwischen dem *Supply*- und dem *Use*-Fall unterschieden werden. Dies bedeutet, daß im *Supply*-

Fall eine Artefaktrepräsentation im Kontext eines CO-Typs Interaktionselemente eines durch diesen CO-Typ angebotenen Interfaces implementiert, während im *Use*-Fall die Interaktionselemente eines genutzten Interfaces implementiert werden. Die Implementierungen der zu Interaktionselementen korrelierten Klassenmethoden der *Composition*-Klassen nutzen wiederum Delegierung (entsprechend [GHJ+ 99]), um für ein Interaktionselement die entsprechende Realisierung im Kontext von Implementierungselementen zu rufen. Die Instanziierung der Artefaktrepräsentationen erfolgt durch Artefaktfabriken (*Artifact Factories*).

Die Implementierungselemente werden - wie bereits diskutiert - durch Methoden der erzeugten Artefaktklassen repräsentiert. Die konkrete Vorschrift zur Erzeugung dieser Klassenmethoden liefert die folgende Regel.

(Regel 54) Für jede einer Instanz des Konzeptes Artefakt zugeordnete Instanz des Konzeptes Implementierungselement wird unter Beachtung der *Requires*- bzw. *Supports*-Relation zwischen implementiertem Interfacetyp im Entwurfsmodell und dem CO-Typ eine mit dem Namen des Implementierungselements bezeichnete Klassenmethode der Artefaktklasse produziert. Die Signatur dieser Klassenmethode ist durch Anwendung von [OMGC++Map] auf die CORBA-IDL-Repräsentation des realisierten Interaktionselements gegeben.
Im Falle von *Continuous-Media*-Interaktionselementen wird für jedes aggregierte Medium im Entwurfsmodell eine Klassenmethode in der Artefaktklasse mit dem Namen **`<Implementation-Element-Name>_<Name der Aggregation>`** erzeugt.
Für *Produce*-Interaktionselemente im *Supports*-Fall bzw. *Consume*-Interaktionselemente im *Requires*-Fall werden die Ableitungsregeln durch *Regel 66* definiert.

Ein Implementierungselement spezifiziert entweder den *Use*- oder *Supply*-Fall in Bezug auf eine *Requires*- oder *Supports*-Relation. In Abhängigkeit vom vorliegenden Fall bedeutet dies für die in der Artefaktklasse zu erzeugenden Methoden die in Tab. 1 dargestellte Methodensemantik zu gewährleisten. Die Parameter dieser

Typ des Interaktionselements im Entwurfsmodell	Falldefinition für das Implementierungs-element im Entwurfsmodell	Semantik der Artefaktmethode
Operation/Attribut	*Supply*-Fall	Implementierung des Operationsverhaltens bzw. Bereitstellung der Zugriffsmethoden für Attribute
Operation/Attribut	*Use*-Fall	nicht erlaubt
Consume	*Supply*-Fall	Implementierung des Konsumierens
Consume	*Use*-Fall	Implementierung des Sendens
Produce	*Supply*-Fall	Implementierung des Sendens
Produce	*Use*-Fall	Implementierung des Konsumierens
Source	*Supply*-Fall	Senden von *Continuous-Media*-Daten
Source	*Use*-Fall	Empfangen von *Continuous-Media*-Daten
Sink	*Supply*-Fall	Empfangen von *Continuous-Media*-Daten
Sink	*Use*-Fall	Senden von *Continuous-Media*-Daten

Tab.1 Semantik der Artefaktmethoden für den Supply- bzw. Use-Fall

Methoden sowie ihr Rückgabetyp ergeben sich kanonisch durch Anwendung von [OMG C++Map] auf die CORBA-IDL-Repräsentation des jeweiligen Interaktionselements.

Das Versenden von Signalen in einer Artefaktklassenrealisierung (*Produce* im *Supports*-Fall, *Consume* im *Requires*-Fall) setzt den Zugang der Artefaktklasseninstanz zu Instanzen von entsprechend *Regel 52* erzeugten Signal-*Producer*-Klassen voraus. Die Instanziierung dieser Klassen geschieht in Verantwortung der CO-Typrepräsentationen und wird später untersucht.

(Beispiel 43) Zur Diskussion der Ableitungsregeln für Implementierungselemente und Artefakte wird die Situation aus *Beispiel 28* um die Definition von Artefakten ***A1***, ***A2*** und ***A3*** und Implementierungselementen im Entwurfsmodell erweitert (vgl. Abb. 24).

- Das Artefakt ***A1*** realisiert das operationale Interaktionselement ***op*** mittels ***a1_op*** für den *Supply*-Fall,
- das Artefakt ***A2*** realisiert die Interaktionselemente ***sendSig*** und ***receiveSig*** mittels der Implementierungselemente ***a2_sig_out*** und ***a2_sig_in*** im *Supply*-Fall und
- das Artefakt ***A3*** realisiert ***sendSig*** und ***receiveSig*** mittels der Implementierungselemente ***a3_sig_in*** und ***a3_sig_out*** im *Use*-Fall.

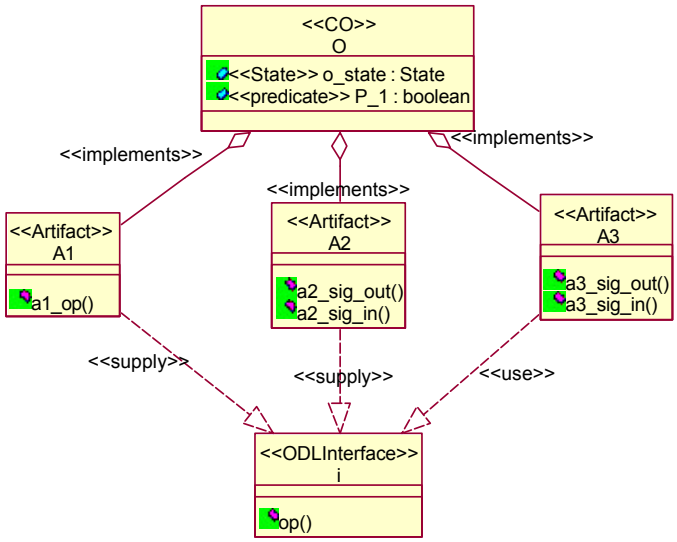


Abb. 24 CO-Typ implementiert durch Artefakte

Für dieses Beispiel implizieren *Regel 53* und *Regel 54* die folgenden Artefakt-Klassendeklarationen:

```
namespace M1
{
 class A1 : public virtual ::Container::Artifact {
 public :
  A1();
  void a1_op ();
  virtual const char * type_name ();
 };

 class A2 : public virtual ::Container::Artifact {
 public :
  A2();
  // Implementierungselement für produziertes Signal,
  // wird definiert durch Regel 66
  void a2_sig_in ( ::M1::Sig * p_a2_sig_in );
  virtual const char * type_name ();
 };
```

```
  class A3 : public virtual ::Container::Artifact {
  public :
    A3();
    void a3_sig_in ( ::M1::Sig * p_a3_sig_in );
    // Implementierungselement für produziertes Signal,
    // wird definiert durch Regel 66
    virtual const char * type_name ();
  };
}
```

Die Bereitstellung von programmiersprachlichen Repräsentationen (hier als Artefaktklassen bezeichnet) für Artefakte, die unabhängig sind von der $CORE_{WARE}$ zugrundeliegenden *Distributed-Processing*-Umgebung CORBA 2.4 als auch unabhängig von einem zu implementierenden Interfacetyp im Entwurfsmodell, sichert die in [CoRE I], Kapitel1 geforderte Flexibilität der Komponierbarkeit von Artefakten, Interfacetypen und CO-Typen. Die Implementierung der Klassenmethoden (*"Business Logic"*) von Artefaktklassen wird entweder durch den Entwickler bereitgestellt oder aber ist durch Verwendung vorgefertigter Implementierungsklassen realisierbar.

2.3.3 Artefaktfabriken und Instanziierungsmuster

$CORE_{WARE}$ muß zur Ausführungszeit Instanzen der oben dargestellten Artefaktklassen erzeugen, um sie an Instanzen einer CO-Typ-*Composition*-Klasse zu binden. Die Instanziierung erfolgt entsprechend den im Entwurfsmodell definierten Instanzen des Konzeptes Instanziierungsmuster.

(Regel 55) Für jedes Artefakt wird eine Artefakt-*Factory*-Klasse mit dem Namen **<Artefaktname>Factory** im C++-Namensraum der Artefaktklasse erzeugt, für die die Methoden **acquire_artifact** und **drop_artifact** definiert sind. Artefakt-*Factory*-Klassen sind als Spezialisierungen der abstrakten Klasse **Container::ArtifactFactory** definiert.

(Beispiel 44) Für das in *Beispiel 43* definierte Artefakt **A1** wird entsprechend *Regel 55* die folgende C++-Klassen für Artefaktfabriken erzeugt:

```
namespace M1 {
  class A1Factory : public virtual ::Container::ArtifactFactory {
  public :
    virtual ::Container::Artifact * acquire_artifact () const;
    virtual void drop_artifact ( ::Container::Artifact * ) const;
    virtual const char * artifact_type_name () const;
  };
}
```

Die Basisklasse **ArtifactFactory** stellt die allgemeine Sicht von $CORE_{WARE}$ auf Artefakt-*Factory*-Klassen dar. Für diese sind abstrakte Methoden zur Anforderung und Freigabe von Artefaktklasseninstanzen definiert. Darüber hinaus ist eine Methode zur Identifizierung des Typs der konkreten Artefaktklasse spezifiziert, der sich aus der Artefaktdefinition im Entwurfsmodell ergibt.

```
namespace Container {
  class ArtifactFactory {
  public :
    virtual Artifact * acquire_artifact () const = 0;
    virtual void drop_artifact ( Artifact * ) const = 0;
    virtual const char * artifact_type_name () const = 0;
  };
}
```

Die Klassenmethoden **acquire_artifact** und **drop_artifact** implementieren die jeweils anzuwendende Instanziierungsregel - die Implementierung wird durch $CORE_{MAP}$ bereitgestellt. Diese Methoden werden durch das Interaktionsmanagement genutzt, um für die Bearbeitung von Interaktionselementen Artefaktklasseninstanzen anzufordern bzw. freizugeben, deren Klassenmethoden gerade die vom Entwickler realisierten Implementierungselemente enthalten.

Die Implementierung der in $CoRE_{CEPT}$ definierten Instanziierungsregeln erfolgt ausschließlich innerhalb der durch $CoRE_{MAP}$ erzeugten Implementierung der konkreten Klassenmethoden **`acquire_artifact`** und **`drop_artifact`**.

(Regel 56) Die Generierung der Implementierung der durch *Regel 55* definierten Klassenmethoden **`acquire_artifact`** und **`drop_artifact`** erfolgt in Abhängigkeit der möglichen Werte von Instanzen des Konzeptes Instanziierungsmuster im vorliegenden Entwurfsmodell:

- für ***`ARTIFACT_PER_REQUEST`*** durch Erzeugen einer Instanz der Artefaktklasse in **`acquire_artifact`** und Zerstören derselben in der Implementierung von **`drop_artifact`**;
- für ***`ARTIFACT_POOL`*** durch Erzeugen einer Menge von Artefaktklasseninstanzen, deren Größe durch Entwurfsmodelldefinitionen vorgegeben ist, während der Konstruktion der Klasse **`<Artefaktname>Factory`**; Entnahme einer Instanz durch **`acquire_artifact`** und Einfügen derselben durch **`drop_artifact`**;
- für ***`SINGLETON`*** durch Erzeugen einer einzigen Artefaktklasseninstanz während der Konstruktion von **`<Artefaktname>Factory`** und Rückgabe derselben durch **`acquire_artifact`** und
- für ***`USER_DEFINED`*** durch Erzeugen von leeren Implementierungen von **`acquire_artifact`** und **`drop_artifact`**.

Die Unterstützung dieser unterschiedlichen Instanziierungsmuster durch die Ableitungsregeln ist die Grundlage der flexiblen Skalierbarkeit von Softwarekomponenten, die mittels $CoRE_{MAP}$ erzeugt werden. Natürlich ist die richtige Auswahl einer Instanziierungsregel die Basis für letztlich skalierbare Softwarekomponenten. Diese Auswahl ist aber anwendungsspezifisch und somit durch den Entwickler vorzunehmen. Jedoch ist die Formulierung „vernünftiger" Richtlinien für die Verwendung von Instanziierungsregeln einem Entwickler einfacher vermittelbar als die konkret anzuwendende Implementierungstechnologie zur Umsetzung derselben. Darüber hinaus kann während der Entwicklung der Softwarekomponenten durch Angabe verschiedener Instanziierungsregeln in Entwurfsmodellen so lange experimentiert werden, bis entsprechende *Performance*-Analysen den Skalierbarkeitsanforderungen an die zu entwickelnde Softwarekomponente gerecht werden.

(Beispiel 45) Exemplarisch sollen hier die Implementierungen der Methoden **`acquire_artifact`** und **`drop_artifact`** für **`::M1::A1Factory`** entsprechend *Beispiel 43* für ***`ARTIFACT_PER_REQUEST`*** und ***`SINGLETON`*** dargestellt werden.

Im Falle ***`ARTIFACT_PER_REQUEST`*** erzeugt die Implementierung von **`acquire_artifact`** eine Instanz von **`::M1::A1`**, die in der Implementierung von **`drop_artifact`** zerstört wird:

```
namespace M1 {
  ::Container::Artifact * A1Factory::acquire_artifact () const {
    return new A1;
  }

  void A1Factory::drop_artifact ( ::Container::Artifact * a ) const {
    delete a;
  }
}
```

Für ***`SINGLETON`*** wird die Klassendeklaration von **`::M1::A1Factory`** um einen leeren Konstruktor, einen virtuellen Destruktor und eine *Private-Member*-Variable vom Typ **`::M1::A1`** erweitert. Während der Konstruktion von **`::M1::A1Factory`** wird die *Member*-Variable mit einer erzeugten Instanz von **`::M1::A1`** belegt, die durch **`acquire_artifact`** zurückgegeben wird. Der Destruktor von **`::M1::A1Factory`** zerstört diese Artefaktklasseninstanz. Aus Gründen der Vereinfachung wurde hier auf die Darstellung der Mechanismen verzichtet, die den parallelen Zugriff (z.B. durch *Threads*) auf diese Instanz durch die Nutzung von *Mutex*-Konstrukten steuern.

```
namespace M1 {
  class A1Factory : public virtual ::Container::ArtifactFactory
  {
    A1 * _instance;
```

```
  public :
    A1Factory ()
      : _instance ( new A1 )
      {}
    virtual ~A1Factory ()
      { delete _instance; }
    virtual ::Container::Artifact * acquire_artifact () const
      { return _instance; }
    virtual void drop_artifact ( ::Container::Artifact * ) const
      {}
    // ...
  };
}
```

Die Implementierung von **artifact_type_name** ist in beiden Fällen gleich und liefert den *Scoped*-Namen des Artefakts im Entwurfsmodell:

```
namespace M1 {
  const char * A1Factory::artifact_type_name () const {
    return "::M1::A1";
  }
}
```

Zur Ausführungszeit muß der Zugriff auf passende Artefakt-*Factory*-Klasseninstanzen durch $CORE_{WARE}$ sichergestellt werden, um letztendlich an *Servant*-Klasseninstanzen eintreffende Operationsrufe via *Composition*-Klasseninstanzen an eine Artefaktklasseninstanz weiterleiten zu können (Korrelation von Interaktions- und Artefaktmanagement). Die Artefaktklasseninstanz muß jedoch zuvor über eine Artefakt-*Factory*-Klasseninstanz angefordert werden. $CORE_{WARE}$ unterstützt diese Funktionalität durch Bereitstellung einer Instanz einer Registrierungsklasse **ArtifactFactoryRegistry**, die wie folgt definiert ist:

```
namespace Container {
  class ArtifactFactoryRegistry
  {
  // ...
  public:
    const ArtifactFactory * resolve_artifact_factory
      ( const char * artifact_type_name );
    void register_artifact_factory
      ( const ArtifactFactory * );
  };
}
```

$CORE_{WARE}$ erzeugt durch den Aufruf einer komponentenspezifischen Initialisierungsfunktion für alle Artefakt-*Factory*-Klassen eine Instanz und registriert diese mittels der oben angegebenen Klassenmethode **register_artifact_factory** an der Instanz der Registrierungsklasse. Die Klassenmethode **resolve_artifact_factory** realisiert das Verhalten zum Auffinden konkreter Artefakt-*Factory*-Klasseninstanzen anhand des *Scoped*-Namens des Artefakts im Entwurfsmodell, die zuvor registriert wurden.

2.3.4 Korrelation zwischen Interaktions- und Artefaktmanagement

Im nächsten Schritt können nun die Ableitungsregeln des Artefaktmanagement mit denen des *Port*- und Interaktionsmanagement kombiniert werden, um das spezifische Verhalten bei Interaktionen durch Implementierungselemente von Artefaktrepräsentationen zu realisieren. Es wird dabei die Behandlung von zu Interfacetypen im Entwurfsmodell zugehörigen Interaktionselementen durch Interface-*Composition*-Klassen untersucht. Darüber hinaus ist die konkrete Implementierung der Interfacenavigation durch die CO-Typ-*Composition*-Klasse zu diskutieren.

2.3.4.1 Interaktionselemente von Interfacetypen

Die Interface-*Composition*-Klassen realisieren das Verhalten für Interaktionselemente, die im Entwurfsmodell im Kontext eines Interfacetyps definiert werden. Dies wird erreicht durch Kooperation der dem Interaktionselement zuzuordnenden Methode der Interface-*Composition*-Klasse mit der Implementierung eines Imple-

mentierungselements im Kontext einer Artefaktrepräsentation. Die Ableitungsregeln, die zur Erzeugung der Interface-*Composition*-Klassen führen, haben die Herstellung des gemeinsamen Kontextes für Interaktionselemente verschiedener Interaktionsarten zum Ziel. Die konkreten Zusammenhänge zwischen Artefaktdefinitionen und Interaktionselementen werden im Entwurfsmodell im Kontext der CO-Typ-Definition berücksichtigt. Diese Definitionen werden durch $CORE_{MAP}$ in den CO-Typ-*Composition*-Klassen repräsentiert, d.h. die Korrelation von Artefakt- und Interaktionsmanagement wird durch Methoden dieser Klassen realisiert. In der Interface-*Composition*-Klasse sind also aus einem Entwurfsmodell nur diejenigen Artefakte bekannt, die die Interaktionselemente des entsprechenden Interfacetyps *potentiell* implementieren, d.h. unabhängig von ihrem Auftreten bei der Definition von CO-Typen, nicht aber die konkret zu verwendenden Artefakte im Kontext eines CO-Typs. Die Lösung dieses Zuordnungsproblems besteht in der Produktion einer abstrakt virtuellen Methode **`resolve_artifact`** innerhalb der Interface-*Composition*-Klasse, die durch die CO-Typ-*Composition*-Klasse als deren Spezialisierung implementiert wird. Damit kann im Kontext einer Interface-*Composition*-Klasse die Methode **`resolve_artifact`** benutzt werden, die erst in der CO-Typ-spezifischen CO-Typ-*Composition*-Klasse implementiert wird.

(Regel 57) Den Interface-*Composition*-Klassen, die entsprechend *Regel 41* produziert wurden, wird die abstrakt virtuelle Methode **`resolve_artifact`** hinzugefügt. Diese Methode hat zwei Parameter vom Typ **`const char *`** bzw. **`ComponentModel::ImplementationDirection`**. Der Rückgabetyp ist **`::Container::Artifact *`**. Zusätzlich definiert jede erzeugte Interface-*Composition*-Klasse eine abstrakt virtuelle Methode **`drop_artifact`** ohne Rückgabetyp, die **`::Container::Artifact *`** als Parameter enthält.

(Beispiel 46) Die Klasse **`iComposition`**, die durch Anwendung von *Regel 57* für den Interfacetyp ***i*** in *Beispiel 28* entsteht, wird gegenüber der Deklaration in *Beispiel 32* folgendermaßen erweitert:

```
namespace ComponentModel {
  enum ImplementationDirection {
    DIR_SUPPLY,
    DIR_USE
  };
}

namespace M1 {
  class iComposition
  {
    // Definitionen wie in Beispiel 32
  public :
    virtual ::Container::Artifact * resolve_artifact
      ( const char * interaction_element,
        ComponentModel::ImplementationDirection direction ) const = 0;
    virtual void drop_artifact ( ::Container::Artifact * ) const = 0;
  };
}
```

Die Methode **`resolve_artifact`** ist so definiert, daß sie für ein mittels des ersten Parameters identifizierbares Interaktionselement und bei Angabe des zu realisierenden *Supply*- bzw. *Use*-Falles eine Instanz einer Artefaktklasse zurückgibt. Das Interaktionselement läßt sich in der Implementierung der Interface-*Composition*-Klasse durch den *Scoped*-Namen desselben im Entwurfsmodell identifizieren. Die Identifikation des **`supply`**- bzw. **`use`**-Falles läßt sich durch die Zugehörigkeit einer Implementierungsmethode zur lokalen Klasse **`__Supply`** oder **`__Use`** feststellen. Der Implementierung der Interface-*Composition*-Klasse ist die Menge aller Repräsentationen von Artefakten im Entwurfsmodell bekannt, die potentiell Interaktionselemente des Interfacetyps implementieren. Insofern kann das Konzept der Reinterpretation (*Dynamic Cast*) des Typs der Artefaktklasseninstanz angewendet werden, um den konkreten Typ der Artefaktklasse zu bestimmen, deren Instanz mittels **`resolve_artifact`** erhalten wurde. Ist der Typ dieser Klasse bekannt, so kann das Implementierungselement im Entwurfsmodell identifiziert werden, das für ein konkretes Interaktionselement die Implementierung des spezifischen Verhaltens übernimmt. Somit kann die entsprechende programmiersprachliche Repräsentation - die Klassenmethode - gerufen werden.

Der Mechanismus der Definition einer abstrakt virtuellen Methodendefinition wird ebenfalls angewendet, um Artefaktklasseninstanzen, die durch die Implementierung der Interface-*Composition*-Klasse angefordert wurden, durch selbige wieder freizugeben. Auch hier wird das konkrete Verhalten durch die CO-Typ-*Composition*-Klasse in Kooperation mit der Artefakt-*Factory*-Klasse erbracht.

Nachdem nunmehr die Identifikation der Artefaktklasseninstanz, an die Aufrufe an *Servant*-Klasseninstanzen delegiert werden, dargestellt wurde, kann die Ableitungsregel von $CORE_{MAP}$ für die Delegierungsmethode an der Interface-*Composition*-Klasse entsprechend definiert werden.

(Regel 58) Für jede Methode der beiden lokalen Klassen **`__Supplier`** und **`__User`** in der Interface-*Composition*-Klasse, die korrespondierend zu Methoden von *Servant*-Klassen entsprechend *Regel 42* erzeugt wurden, wird eine Implementierung produziert. Diese Implementierung bestimmt unter Benutzung der durch *Regel 57* definierten Klassenmethode **`resolve_artifact`** die Artefaktklasseninstanz, an die entsprechend den Festlegungen im Entwurfsmodell delegiert werden muß. In der Implementierung der Delegationsmethode muß dynamisch der Typ der Artefaktklasse ermittelt werden, zu der die Artefaktklasseninstanz gehört. Dies wird durch eine Sequenz von **`dynamic_cast`**-Operationsrufen realisiert. Sofern die entsprechende Artefaktklasse identifiziert wurde, wird die durch diese Identifizierung implizierte Methode als Repräsentation des Implementierungselements (*Regel 54*) an der Artefaktklasseninstanz gerufen. Anschließend wird für die genutzte Artefaktklasseninstanz **`drop_artifact`** ausgeführt. Falls die Klasseninstanz nicht identifizierbar ist, wird die CORBA-Ausnahme **`CORBA::NO_IMPLEMENT`** ausgelöst.

(Beispiel 47) Die Implementierung der Klassenmethode **`iComposition::__Supplier::op`**, die das Interaktionselement ***`op`*** des Interfacetyps ***`i`*** im *Supply*-Fall entsprechend *Beispiel 43* behandelt, ist durch Anwendung von *Regel 58* gegeben durch:

```
namespace M1{
 void iComposition::__Supplier::op
 ( const PortableServer::ServantBase * servant ) const
 {
  ::Container::Artifact * artifact_ = this->_composition->resolve_artifact
    ( "::M1::i::op", ComponentModel::DIR_SUPPLY );
  if ( dynamic_cast < ::M1::A1 * > ( artifact_ ) )
    ( dynamic_cast < ::M1::A1 * > ( artifact_ ) ) -> a1_op ();
  else
    throw CORBA::NO_IMPLEMENT();
  this->_composition->drop_artifact ( artifact_ );
 }
}
```

Verändert man die Situation insofern, als das für das Interaktionselement ***`op`*** ein weiteres Implementierungselement ***`ax_op`*** im Kontext eines zusätzlichen Artefakts ***`Ax`*** im Entwurfsmodell definiert wird, so verändert sich gemäß *Regel 58* die Implementierung von **`iComposition::__Supplier::op`** wie folgt:

```
namespace M1 {
 void iComposition::__Supplier::op
 ( const PortableServer::ServantBase * servant ) const
 {
  ::Container::Artifact * artifact_ = this->_composition->resolve_artifact
    ( "::M1::i::op", ComponentModel::DIR_SUPPLY );
  if ( dynamic_cast < ::M1::A1 * > ( artifact_ ) )
    ( dynamic_cast < ::M1::A1 * > ( artifact_ ) ) -> a1_op ();
  else if ( dynamic_cast < ::M1::A2 * > ( artifact_ ) )
    ( dynamic_cast < ::M1::A2 * > ( artifact_ ) ) -> ax_op ();
  else
    throw CORBA::NO_IMPLEMENT();
  this->_composition->drop_artifact ( artifact_ );
 }
}
```

Offen ist bisher noch die Definition des Verhaltens der Interface-*Composition*-Klassenimplementierung, falls im Entwurfsmodell keine Implementierungselemente für einen Interfacetyp angegeben sind, der durch diese

Klasse repräsentiert wird. Durch $CORE_{MAP}$ wird in diesem Fall aus Gründen der Vereinfachung der prototypischen Realisierung im Kontext der Implementierung der Interface-*Composition*-Klasse die CORBA-Systemausnahme **CORBA::NO_IMPLEMENT** ausgelöst.

(Regel 59) Falls in einem Entwurfsmodell keine realisierende Instanz des Konzeptes Implementierungselement für eine Instanz des Konzeptes Interaktionselement angegeben ist, so wird durch die Implementierung der Interface-*Composition*-Klasse die Ausnahme **CORBA::NO_IMPLEMENT** ausgelöst.

Durch *Regel 57* bis *Regel 59* ist nun das Verhalten der Interface-*Composition*-Klassen bei der Behandlung von Interaktionselementen inklusive des korrekten Aufrufes der für assoziierte Implementierungselemente erzeugten Klassenmethoden von Artefaktrepräsentationen definiert. Die nachfolgende Untersuchung stellt nun die Realisierung von Artefakt- und Interaktionsmanagement in der CO-Typ-*Composition*-Klasse durch $CORE_{MAP}$ dar.

2.3.4.2 Integration von Artefakt- und Interaktionsmanagement

Die CO-Typ-*Composition*-Klasse stellt den gemeinsamen Kontext aller in Interfacetypen definierten Interaktionselemente her. Dazu wird das *Port*-Management mit dem Interaktions- und Artefaktmanagement integriert. Diese Integration erfolgt durch die Realisierung der CORBA-IDL-Interfacedefinition **::ComponentModel::ComponentBase** spezifisch für jeden CO-Typ.

Entsprechend der Definition von *Regel 22* realisiert jede CO-Typrepräsentation die CORBA-IDL-Interfacedefinition **::ComponentModel::ComponentBase** an. Diese CORBA-IDL-Interfacedefinition wird durch die Klasse **ComponentModel::ComponentBaseImpl** - bereitgestellt durch $CORE_{WARE}$ - realisiert. Diese Basisklasse implementiert zum einen die CORBA-IDL-Interfacedefinition **ComponentBase**, zum anderen stellt sie für CO-Typ-Repräsentationen allgemeine Funktionalität zur Interfacenavigation und Reflexion bereit. Neben dieser Basisklasse muß eine endgültige CO-Typ-Repräsentation dem Entwickler Möglichkeiten bieten, unabhängig von der Behandlung von Interaktionselementen in Artefaktrepräsentationen - gesteuert durch die CO-Typ-*Composition*-Klasse - Interaktionen und Aktionen auszulösen. Dies ist beispielsweise zur Ansteuerung einer graphischen Nutzeroberfläche oder zum Initiieren von Interaktionen mit anderen COs notwendig. Die folgende Ableitungsregel definiert für CO-Typen eine produzierte C++-Klasse, die mittels Vererbung die Funktionalität der Klasse **ComponentModel::ComponentBaseImpl** bereitstellt sowie definiert, auf welche Weise ein Entwickler unabhängig von der Reaktion auf Interaktionselemente Aktionen bzw. Interaktionen auslösen kann.

(Regel 60) Zur Repräsentation eines CO-Typs in $CORE_{WARE}$ wird eine C++-Klasse mit dem Namen **CO_<Name des CO-Typs>** erzeugt. Diese spezialisiert die entsprechend *Regel 44* für den CO-Typ erzeugte CO-Typ-*Composition*-Klasse und die Klasse **ComponentModel::ComponentBaseImpl**.

Die Vererbungshierarchie, die sich durch Anwendung von *Regel 60* für CO-Typrepräsentationen ergibt, ist in Abb. 25 dargestellt. Entsprechend *Regel 44* wird eine Klasse **<CO-Typ>Composition** (für einen CO-Typ ***CO_model*** entsprechend **CO_modelComposition**) erzeugt, die alle Interface-*Composition*-Klassen spezialisiert (in der Abbildung **I1_Composition** bis **In_Composition**), zu denen der CO-Typ im Entwurfsmodell entweder eine *Requires*- oder eine *Supports*-Relation definiert. Die entsprechend *Regel 60* definierte Klasse **CO_<CO-Typ>** leitet nun von der für den CO-Typ erzeugten CO-Typ-*Composition*-Klasse und der durch die *Component-Support*-Plattform definierten Klasse **ComponentBaseImpl** ab.

```
namespace ComponentModel {
  class ComponentBaseImpl {
    // ...
  public :
    // ...
    virtual ComponentKey* primary_key();
    COFactoryBaseImpl * creator ();
  };
}
```

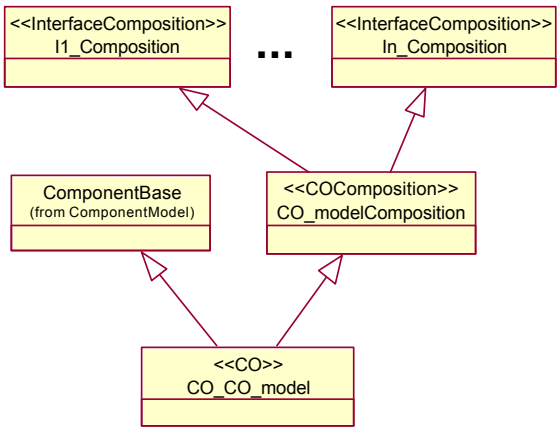


Abb. 25 Vererbungshierarchie für CO-Typ-Repräsentationen

Die Definition von **`ComponentBaseImpl`** zeigt, daß die Realisierung der CO-Identität Bestandteil von *CORE*$_{WARE}$ ist, diese Funktionalität also durch Spezialisierung dieser Klasse für konkrete CO-Typen nutzbar gemacht wird. Eine Identifizierung eines COs ist für die Umgebung dieses CO durch Aufruf der Methode **`primary_key`** möglich, die die Operation **`primary_key`** der CORBA-IDL-Interfacedefinition **`ComponentBase`** implementiert.

Um einem Entwickler die Möglichkeit zu geben, zu beliebigen Zeitpunkten (d.h. nicht ausschließlich als Reaktion auf Interaktionen) Interaktionen zu initiieren oder eine CO-Typrealisierung mit einem graphischen Nutzerinterface zu integrieren, definiert die Klasse **`CO_<CO-Typ>`** eine Klassenmethode **`main`** ohne Argumente und ohne Rückgabetyp. Diese Methode wird zur Ausführungszeit dann gerufen und nebenläufig ausgeführt, wenn eine Instanz der Klasse **`CO_<CO-Typ>`** gebildet wird, falls ein CO instanziiert wird.

(Regel 61) Die entsprechend *Regel 60* erzeugte C++-Klasse, die den CO-Typ repräsentiert, erhält eine zusätzliche Methode **`main`**, deren Implementierung durch einen Entwickler anwendungsspezifisch vorgenommen werden kann. Diese Methode wird nach der Instanziierung der den CO-Typ repräsentierenden Klasse gerufen und nebenläufig ausgeführt.

Durch die Ableitungsregeln von *CORE*$_{MAP}$ ist sichergestellt, daß Operationsrufe an *Servant*-Klasseninstanzen, die für CORBA-IDL-Interfacedefinitionen erzeugt wurden, an eine Instanz der entsprechend *Regel 61* resultierenden Klasse **`CO_<Name des CO-Typs>`** delegiert werden.

Dementsprechend kann die konkrete Korrelation von Interaktions- und Artefaktmanagement durch die Klasse **`CO_<CO-Typ>`** realisiert werden mittels Bereitstellung einer CO-Typ-spezifischen Implementierung der durch Vererbung definierten virtuellen Methode **`resolve_artifact`**, die durch die spezialisierten Interface-*Composition*-Klassen abstrakt deklariert wurde (vgl. *Regel 57*).

(Regel 62) Die Klasse **`CO_<CO-Typ>`** implementiert die entsprechend *Regel 57* erzeugte virtuelle Methode **`resolve_artifact`**. Die Implementierung wird unter Berücksichtigung des Namens des Interaktionselements im Entwurfsmodell sowie des *Use*- bzw. *Supply*-Falles vorgenommen. Die Implementierung erfolgt in den folgenden Schritten:

- vergleiche, ob der im ersten Parameter übergebene Name einem Namen eines Interaktionselements eines Interfacetyps entspricht, zu dem eine *Supports*- oder *Requires*-Relation im Entwurfsmodell definiert ist (Beachtung des *Use*- bzw. *Supply*-Falles),
- falls der Vergleich für alle Interaktionselemente fehlschlägt, gebe den Wert 0 zurück,
- falls der Vergleich erfolgreich ist, löse die zugehörige Artefakt-*Factory*-Klasseninstanz auf und erzeuge eine Artefaktinstanz,

- übergebe der angelegten Artefaktklasseninstanz den **this**-Zeiger und
- gebe die erzeugte Artefaktklasseninstanz zurück.

Durch *Regel 62* ist nicht nur die Kopplung von Artefakt- und Interaktionsmanagement definiert, sondern auch, auf welche Weise die Implementierung eines Implementierungselements in einer Artefaktrepräsentation aktuelle, mit einem CO verbundene Zustandsinformationen auflösen kann.

(Beispiel 48) Für die in *Beispiel 28* dargestellte Situation ergibt sich durch Anwendung von *Regel 62* die folgende, auszugsweise präsentierte Implementierung für **::M1::CO_O::resolve_artifact**:

```
namespace M1 {
 ::Container::Artifact * CO_O::resolve_artifact
  ( const char * interaction_element,
    ComponentModel::ImplementationDirection direction ) const
 {
  const ::Container::ArtifactFactory * artifact_factory_;
  // Vergleich des übergebenen Namens mit den Interaktionselementen unter
  // Beachtung des supply- bzw. use-Falls
  if (  strcmp ( interaction_element, "::M1::i::op" ) == 0
     && direction == ComponentModel::DIR_SUPPLY )
  {
   // Auflösen der Artifact-Factory-Klasseninstanz
   artifact_factory_ =
     (::Container::Container::instance()
      ->get_artifact_registry()).resolve_artifact_factory ( "::M1::A1" );
   // Nutzung der aufgelösten Artifact-Factory-Klasseninstanz zur Instanzierung
   // eines Artifacts, vgl. Regel 55
   ::Container::Artifact * ret_ = artifact_factory_->acquire_artifact();
   ret_ -> factory ( artifact_factory_ );
   // Hinterlegung des CO-Kontextes durch Referenz auf this
   ret_ -> co ( dynamic_cast < const ::ComponentModel::ComponentBaseImpl* > (this));
   // Rückgabe der Artefaktklasseninstanz
   return ret_;
  }
  else if ( strcmp ( interaction_element, "::M1::i::sendSig" ) == 0
         && direction == ComponentModel::DIR_SUPPLY )
  {
   // ...
  }
  else if ( strcmp ( interaction_element, "::M1::i::receiveSig" ) == 0
         && direction == ComponentModel::DIR_SUPPLY )
  {
   // ...
  }
  else if ( strcmp ( interaction_element, "::M1::i::sendSig" ) == 0
         && direction == ComponentModel::DIR_USE )
  {
   // ...
  }
  else if ( strcmp ( interaction_element, "::M1::i::receiveSig" ) == 0
         && direction == ComponentModel::DIR_USE )
  {
   // ...
  }
  // Vergleich schlug fehl, Rückgabe von 0
  return 0;
 }
}
```

Durch die beschriebenen Ableitungsregeln von $CORE_{MAP}$ ist nunmehr die Anbindung der für Interfacetypen im Entwurfsmodell generierten *Servant*-Klassen an die Repräsentationen der Artefaktklassen entsprechend der im Entwurfsmodell spezifizierten Implementierungselemente vollständig möglich. Die externen Interaktionspunkte von COs sind mittels Interaktions- und Artefaktmanagement an die durch den Entwickler bereitgestellten Realisierungen der im Entwurfsmodell definierten Implementierungselemente angebunden.

Neben der Kopplung von Interaktions- und Artefaktmanagement ist die Realisierung der Interfacenavigation innerhalb der Implementierung der CO-Typ-Repräsentation zu untersuchen. Wie bereits diskutiert, wird die Interfacenavigation vollständig durch von $CoRE_{MAP}$ erzeugte Implementierungen behandelt. Die dabei angewendeten Mechanismen werden im folgenden vorgestellt.

2.3.4.3 Interfacenavigation

Die Interfacenavigation wird durch die entsprechend *Regel 46* bzw. *Regel 47* produzierten Methoden **provide_<Port-Name>**, **provide_<Port-Name>_supply** (**provide**-Operationen) und **connect_<Port-Name>** der Klasse **<CO-Typ>Composition** implementiert. Für die **provide**-Operationen zerfällt die Realisierung dabei in den operationalen und nicht-operationalen Anteil. Im ersten Fall wird nur eine Referenz des CORBA-Interfaces zurückgegeben, das den operationalen Teil eines Interfacetyps im Entwurfsmodell repräsentiert. Da die Klasse **<CO-Typ>Composition** selbst eine *Servant*-Klasseninstanz erzeugt, die diese CORBA-IDL-Interfacedefinition implementiert, kann eine entsprechende CORBA-Objektreferenz dieser *Servant*-Klasseninstanz einfach ermittelt werden.

(Regel 63) Die Klassenmethode **provide_<Port-Name>**, die entsprechend *Regel 45* im Kontext der Klasse **<CO-Typ>Composition** produziert wurde, wird durch Rückgabe des Wertes von **_<Port-Name>->_this()** implementiert.

Mittels der Klassenmethode **_this()** der zu einer *Port*-Definition gehörigen *Servant*-Klasse wird eine CORBA-Interfacereferenz generiert. Diese Referenz verweist auf ein CORBA-Interface, das den operationalen Anteil des Interfacetyps im Entwurfsmodell repräsentiert, der an dieser *Port*-Definition angeboten wird. Die *Servant*-Klasseninstanz wurde entsprechend *Regel 45* während der Konstruktion der Klasse **<CO-Typ>Composition** erzeugt und in der *Member*-Variablen **_<Port-Name>** hinterlegt.

(Beispiel 49) Die Implementierung der Klassenmethode **provide_serve** für die mit ***serve*** bezeichnete *Port*-Definition aus *Beispiel 28* ergibt sich entsprechend *Regel 63*:

```
namespace M1 {
  i_ptr OComposition::provide_serve
  ( const PortableServer::ServantBase * servant ) const
  {
    return _serve -> _this();
  }
}
```

Während die Implementierung der **provide**-Methode für den operationalen Anteil eines Interfacetyps relativ simpel ist, fällt die Realisierung für den nicht-operationalen Anteil ungleich komplexer aus. Der Grund dafür besteht in der Konfiguration der durch $CoRE_{WARE}$ bereitgestellten Kommunikationskanäle für die Realisierung von Signal- und *Continuous-Media*-Interaktionselementen. Diese Konfiguration erfolgt durch die produzierte Implementierung für **provide_<Port-Name>_supply**.

Zur Präzisierung dieser Konfigurationsaspekte ist es zunächst notwendig, die Bildung von Instanzen der durch *Regel 52* definierten Signal-*Producer*-Klassen zu diskutieren, die der Artefaktrealisierung zum Versenden von Signalen zur Verfügung stehen. Diese Klassen implementieren die lokalen CORBA-IDL-Interfacedefinitionen, die durch die Ableitungsregeln der Struktursicht für den Produzenten eines Signals erzeugt werden. Sie stellen Vermittler dar, denen die Repräsentation eines Signals übergeben wird, und die nachfolgend einen durch $CoRE_{WARE}$ bereitgestellten Mechanismus nutzen, um das Signal an einen Empfänger zu übertragen. Es ist also einerseits zu untersuchen, wie und zu welchem Zeitpunkt diese Klassen instanziiert werden. Andererseits ist bisher offen, wie eine Artefaktklasseninstanz, die das Senden von Signalen implementiert, Zugang zu einer benötigten Signal-*Producer*-Klasseninstanz erhält.

Das Management der Instanzen aller Signal-*Producer*-Klassen, die im Kontext eines Interfacetyps im Entwurfsmodell unter Beachtung der *Supply*- bzw. *Use*-Fälle durch Regel *Regel 52* erzeugt werden, wird durch eine von $CoRE_{MAP}$ produzierte *Supplier*-Klasse realisiert. Diese Klasse implementiert die Kopplung der Signal-*Producer*-Klasseninstanzen an den Signalkommunikationsmechanismus von $CoRE_{WARE}$.

(Regel 64) Für das Management der durch *Regel 52* erzeugten Signal-*Producer*-Klassen werden im Kontext eines Interfacetyps im Entwurfsmodell zwei Klassen mit dem Namen **<Interfacename>__Supplier_Impl** in den Namensräumen **<Interfacename>__Supply__** für die Bereitstellung und **<Interfacename>__Use__** für die Nutzung dieses Interfacetyps erzeugt. Diese Klassen definieren private *Member*-Variablen mit dem Typ der entsprechend *Regel 52* im gleichen Namensraum produzierten Signal-*Producer*-Klassen sowie entsprechende Zugriffsmethoden. Diese *Member*-Variablen werden im Konstruktor der Klasse **<Interfacename>__Supplier_Impl** initialisiert und mit Kommunikationskanälen für Signalinteraktionselemente von *CORE*$_{WARE}$ verbunden. Der Destruktor der Klasse **<Interfacename>__Supplier_Impl** entkoppelt diese Instanzen von den Kommunikationskanälen durch Aufruf der Destruktoren.

Diese Regel läßt zunächst die konkrete Implementierung des Konstruktors der Klasse **<Interfacename>__Supplier_Impl** offen, die für die Realisierung der Anbindung aller Signal-*Producer*-Klasseninstanzen an die Signalkommunikationskanäle von *CORE*$_{WARE}$ verantwortlich ist. Die Erzeugung dieser Implementierung wird im weiteren definiert.

(Beispiel 50) Durch *Regel 64* wird für den Anbieter eines Interfacetyps ***i*** aus *Beispiel 28* die folgende Klasse erzeugt, die für das Management der Signal-*Producer*-Klasseninstanz für das *Produce*-Interaktionselement ***sendSig*** zuständig ist:

```
namespace M1 {
 namespace i__Supply__ {
  class i__Supplier_Impl : public Container::LocalObjectImpl
  {
   sendSig_Producer * _sendSig_supplier;
  public :
   i__Supplier_Impl
    ( const ComponentModel::ComponentBaseImpl *, ::ComponentModel::Parameters& );
   virtual ~i__Supplier_Impl()
    {
    // Abbau aller angeforderten CORBA-Event-Kanäle durch Zerstören der
    // SignalProducer-Instanzen
    if ( _sendSig_supplier ) delete _sendSig_supplier;
    }
   sendSig_Producer * sendSig_supplier()
    { return _sendSig_supplier; }
  };
 }
}
```

Die Kopplung der Signal-*Producer*-Klasseninstanzen an die Signalkommunikationskanäle wird während der Konstruktion der *Supplier*-Klasse vorgenommen.

(Regel 65) Dem Konstruktor der entsprechend *Regel 65* produzierten *Supplier*-Klasse werden die Instanz der korrespondierenden CO-Typrepräsentation **const ComponentModel::ComponentBaseImpl *** und eine **inout**-Liste von Parametern übergeben (vgl. *Regel 28*). Für alle Interaktionselemente des Typs *Produce* im *Supply*-Fall und *Consume* im *Use*-Fall werden die folgenden Aktionen ausgeführt:

- falls die Parameter eine Instanz von **ChannelIdentification** für das Interaktionselement enthalten, wird der dadurch identifizierte CORBA-*Event*-Kanal aufgelöst,
- anderenfalls wird für das Interaktionselement ein CORBA-*Event*-Kanal erzeugt und dessen Identifizierung als neuer **ChannelIdentification**-Parameter in der Liste der Parameter verzeichnet,
- abschließend wird eine Instanz der korrespondierenden Signal-*Producer*-Klasse angelegt und dieser der aufgelöste bzw. erzeugte CORBA-*Event*-Kanal als Konstruktionsparameter übergeben.

Für ein einzelnes Signalinteraktionselement ist der schematische Ablauf dieser Kopplung in Abb. 26 dargestellt. Dieser Ablauf wiederholt sich im *Supply*-Fall für alle *Produce*-Interaktionselemente, im *Use*-Fall entsprechend für alle *Consume*-Interaktionselemente. *CORE*$_{WARE}$ bietet geeignete Interfaces zum Auflösen bzw.

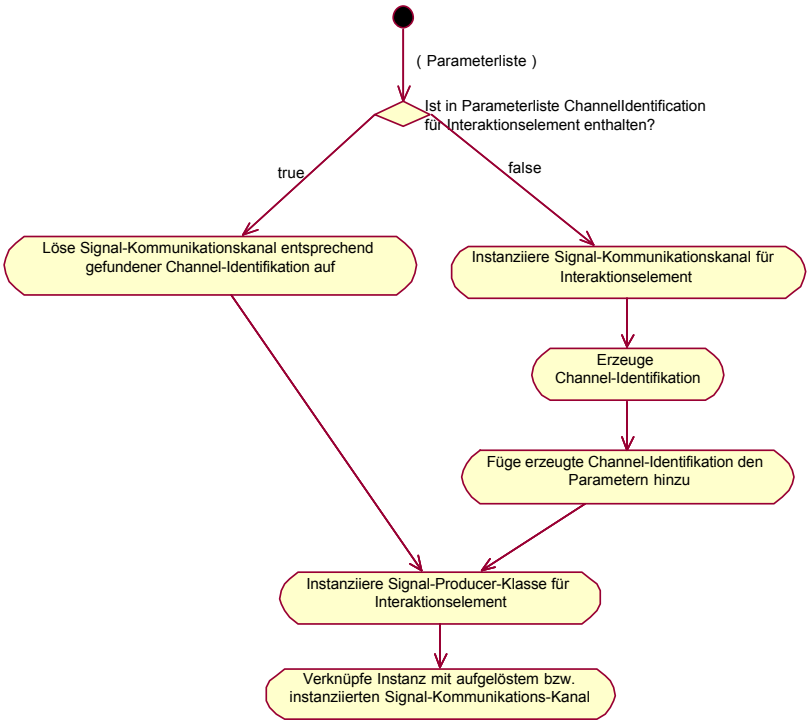


Abb. 26 Kopplung von Signal-Producer-Klasseninstanzen an Signalkommunikationskanäle der Ausführungsumgebung

Erzeugen von Signal-Kommunikationskanälen an. Darüber hinaus implementiert sie einen Erzeugungsmechanismus für Kanalidentifikationsparameter.

(Beispiel 51) Der Konstruktor der Klasse **`i__Supplier_Impl`** wird entsprechend *Regel 65* folgendermaßen realisiert:

```
namespace M1 {
 namespace i__Supply__ {
  i__Supplier_Impl::i__Supplier_Impl
  ( const ComponentModel::ComponentBaseImpl * co,
    ::ComponentModel::Parameters& params )
   : Container::LocalObjectImpl ( co )
  {
   // Überprüfe, ob für produce-Interaktionselement bereits ein Event-Kanal
   // erzeugt wurde
   CosEventChannelAdmin::EventChannel_var channel_
    = ::Container::Container::instance()->find_channel ( params, "sendSig" );
   // falls nicht, instanziiere Event-Kanal
   if ( CORBA::is_nil ( channel_ ) )
   {
    // Generiere Identifikation vom Typ UUID
    // uuid_ = ...
    // Instanziere CORBA-Event-Kanal mit generierter Identifikation
    // channel_ = ...
    // generiere Konfigurationsparameter vom Typ ChannelIdentification mit den
    // Parametern uuid_ zur Channel-Identifikation, Name des Interaktionselements
    // und Referenz auf CosEventAdmin::EventChannelAdmin-Objekt
    ComponentModel::Container::ChannelIdentification* channel_id_
     = Container::ChannelIdentificationFactoryImpl::instance()->create
       ( uuid_, "sendSig", channel_ );
    // füge generierten Parameter der inout-Parameter-Liste hinzu
    params.length ( params.length() + 1 );
```

```
        params [ params.length() - 1 ] = channel_id_;
       }
       if ( ! CORBA::is_nil ( channel_ ) )
        _sendSig_supplier = new sendSig_Producer ( co, channel_ );
       else
        // Fehlerbehandlung
      }
     }
    }
```

Durch die Erzeugung von Signal-*Producer*-Klasseninstanzen durch Instanzen der *Supplier*-Klassen ist nun die korrekte Anbindung dieser Signalproduzenten an die Signalkommunikationskanäle von $CORE_{WARE}$ sichergestellt. Um der Realisierung der Implementierungselemente im Kontext einer Artefaktrepräsentation den Zugriff auf die instanziierten Signal-*Producer*-Klasseninstanzen zu erlauben, werden den Artefaktrepräsentationen weitere Klassenmethoden hinzugefügt.

(Regel 66) Für Implementierungselemente, die auf *Produce*- im *Supply*-Fall bzw. *Consume*-Interaktionselemente im *Use*-Fall verweisen, werden im Kontext von Artefaktrepräsentationen Klassenmethoden mit dem Namen **notify_connect_<Implementierungselements>** ohne Rückgabetyp erzeugt. Diese Methoden erhalten einen Zeiger auf eine entsprechend *Regel 52* produzierte Signal-*Producer*-Klasse als Parameter.

(Beispiel 52) Im Kontext der Repräsentation des Artefakts **A2** aus *Beispiel 28* wird entsprechend *Regel 66* die Methode **notify_connect_a2_sig_out** deklariert:

```
namespace M1
{
 class A2 : public virtual ::Container::Artifact
 {
 public :
  // ...
  void notify_connect_a2_sig_out ( ::M1::i__Supply__::sendSig_Producer * );
 // ...
 };
}
```

Mit Hilfe von *Regel 66* hat die Artefaktrealisierung Zugriff auf die benötigte Signal-*Producer*-Klasseninstanz, die zum (getypten) Versenden von Signalen genutzt wird. Durch die dementsprechende Vervollständigung der Ableitungsregeln für Artefakte kann im folgenden die Realisierung der **provide**-Methode für den nicht-operationalen Anteil dargestellt werden.

(Regel 67) Die Methode **provide_<Port-Name>_supply**, die entsprechend *Regel 45* im Kontext der Klasse **<CO-Typ>Composition** erzeugt wurde, implementiert die Konfiguration der Kommunikationskanäle für Signal- und *Continuous-Media*-Interaktionselemente und gibt den Wert von **_<Port-Name>_supply->_this()** zurück.
Die Konfiguration von *Continuous-Media*-Kommunikationskanälen wird durch $CORE_{WARE}$ unter Nutzung der CORBA-Objektreferenzen vom Typ **::ComponentModel::MediaManager** vorgenommen, auf die mittels der CORBA-Objektreferenz vom Typ **<Interfacename>__Supply** zugegriffen werden kann.
Die Konfiguration der Kommunikationskanäle für *Produce*- bzw. *Consume*-Interaktionselemente wird entsprechend folgendem Schema vorgenommen:

- erzeuge eine Instanz der durch *Regel 64* produzierten Klasse **<Interfacename>__Supply__::<Interfacename>__Supplier_Impl**,
- löse für jedes *Produce*-Interaktionselement des Interfacetyps im Entwurfsmodell mit dem durch *Regel 62* definierten Mechanismus die implementierende Artefaktklasseninstanz auf und rufe die Methode **notify_connect_<Name des Implementierungselements>** an dieser Instanz,
- löse für jedes *Consume*-Interaktionselement einen CORBA-*Event*-Kanal von $CORE_{WARE}$ auf, falls in der Liste der Parameter identifiziert, oder instanziiere einen Kanal (analog zu *Regel 65*),

- verbinde die CORBA-Objektreferenz, die der *Servant*-Klasseninstanz **<Interfacename>__Supply** zugeordnet ist, mit dem aufgelösten bzw. erzeugten CORBA-*Event*-Kanal von *CORE*$_{WARE}$ und
- gebe die CORBA-Objektreferenz zurück, die der *Servant*-Klasseninstanz **<Interfacename>__Supply** in der *Member*-Variablen **_<Port-Name>_supply** zugeordnet ist.

Das durch *Regel 67* definierte Verfahren zur Kopplung von Signalkommunikationskanälen folgt dem in Abb.26 präsentierten Ablauf.

(Beispiel 53) Die Implementierung der Klassenmethode **provide_serve_supply** für die mit ***serve*** bezeichnete *Port*-Definition aus *Beispiel 28* ergibt sich entsprechend *Regel 67*:

```
namespace M1 {
 ::M1::i__Supply__::i__Supply_ptr OComposition::provide_serve__Supply
 ( ::ComponentModel::Parameters& params,
  const PortableServer::ServantBase * servant ) const
 {
  i__Supply__::i__Supplier_Impl * supplier_
   = new i__Supply__::i__Supplier_Impl
    ( dynamic_cast < const ComponentModel::ComponentBaseImpl * > ( this ) , params );
  ::Container::Artifact * artifact_ = 0;
  artifact_ = this->resolve_artifact ( "::M1::i::sendSig", ComponentModel::DIR_SUPPLY );
  if ( dynamic_cast < ::M1::A2 * > ( artifact_ ) )
   dynamic_cast < ::M1::A2 * > ( artifact_ )
    -> notify_connect_a2_sig_out ( supplier_ -> sendSig_supplier() );
  else
   throw CORBA::NO_IMPLEMENT();
   // Überprüfe, ob für consume-Interaktionselement bereits ein Event-Kanal
   // erzeugt wurde
  CosEventChannelAdmin::EventChannel_var channel_
   = ::Container::Container::instance()->find_channel ( params, "receiveSig" );
   // falls nicht, instanziiere Event-Kanal
  if ( CORBA::is_nil ( channel_ ) )
  {
   // Generiere Identifikation vom Typ UUID
   // uuid_ = ...
   // Instanziiere CORBA-Event-Kanal mit generierter Identifikation
   // channel_ = ...
   // generiere Konfigurationsparameter vom Typ ChannelIdentification mit den
   // Parametern uuid_ zur Channel-Identifikation, Name des Interaktionselements
   // und Referenz auf CosEventAdmin::EventChannelAdmin-Objekt
   CORBA::ValueBase * base_
    = Container::ChannelIdentificationFactoryImpl::instance()->create
     ( uuid_, "receiveSig", channel_ );
   ComponentModel::Container::ChannelIdentification* channel_id_
    = ComponentModel::Container::ChannelIdentification::_downcast ( base_ );
   // füge generierten Parameter der inout-Parameter-Liste hinzu
   params.length ( params.length() + 1 );
   params [ params.length() - 1 ] = channel_id_;
  }
  try
  {
   // verbinde _serve_supply mit aufgelöstem/instanziiertem Event-Kanal
   CosEventChannelAdmin::ProxyPushSupplier_var proxy_push_supplier_
    = channel_->for_consumers()->obtain_push_supplier();
   proxy_push_supplier_->connect_push_consumer ( this -> _serve_supply->_this() );
  }
  catch ( CORBA::Exception& )
  {
   // Fehlerbehandlung
  }
  return _serve_supply -> _this();
 }
}
```

Die Definition dieser Regel vervollständigt die Implementierung der Interfacenavigation für angebotene Interfacetypen. Insbesondere wird deutlich, daß Interfacenavigation ausschließlich durch die produzierten Klassen der Repräsentation von CO-Typen realisiert werden kann. Nun lassen sich die gleichen Mechanismen auch zur Realisierung der Interfacenavigation für genutzte Interfacetypen anwenden.

(Regel 68) Die Implementierung der durch *Regel 46* produzierten Klassenmethode **connect_<Port-Name>** der Klasse **<CO-Typ>Composition** erfolgt in Analogie zu **provide_<Port-Name>_supply** entsprechend folgendem Schema:

- hinterlege die im Parameter **<Port-Name>** übergebene CORBA-Objektreferenz für den operationalen Anteil des genutzten Interfacetyps mittels der Klassemethode **set_<Port-Name>**,
- erzeuge eine Instanz der durch *Regel 64* produzierten Klasse **<Interfacename>__Use__::<Interfacename>__Supplier_Impl**,
- löse für jedes *Consume*-Interaktionselement des Interfacetyps im Entwurfsmodell mit dem durch *Regel 62* definierten Mechanismus die implementierende Artefaktklasseninstanz auf und rufe die Methode **notify_connect_<Name des Implementierungselements>** an dieser Instanz,
- löse für jedes *Produce*-Interaktionselement einen CORBA-*Event*-Kanal von $CORE_{WARE}$ auf, falls in der Liste der Parameter identifiziert, oder instanziiere einen Kanal (analog zu *Regel 65*),
- verbinde die CORBA-Objektreferenz, die der *Servant*-Klasseninstanz **<Interfacename>__Supply** zugeordnet ist, mit dem aufgelösten bzw. erzeugten CORBA-*Event*-Kanal von $CORE_{WARE}$ und
- gebe die CORBA-Objektreferenz zurück, die der *Servant*-Klasseninstanz **<Interfacename>__Use** in der *Member*-Variablen **_<Port-Name>_use** zugeordnet ist.

(Beispiel 54) Für die **connect**-Methode der Klasse **::M1::OComposition** aus *Beispiel 28* ergibt sich entsprechend *Regel 68* die folgende Implementierung:

```
namespace M1 {
 ::M1::i__Use__::i__Use_ptr  OComposition::connect_use
 ( ::M1::i_ptr use,
  ::ComponentModel::Parameters& params,
  const PortableServer::ServantBase * servant ) const
 {
  // hintelege Objektreferenz für operationalen Anteil von i mittels set_use
  ( ( OComposition* ) this ) -> set_use ( use );
  // erzeuge Instanz der Klasse i__Use__::Supplier_Impl zum Management der Signal-
  // Producer-Klasseninstanz für receiveSig
  ::M1::i__Use__::i__Supplier_Impl * supplier_
   = new ::M1::i__Use__::i__Supplier_Impl
    ( dynamic_cast < const ComponentModel::ComponentBaseImpl * > ( this ) , params );
  // Löse die Artefaktinstanz für receiveSig auf und benachrichtige diese Instanz
  ::Container::Artifact * artifact_
   = this->resolve_artifact ( "::M1::i::receiveSig", ComponentModel::DIR_USE );
  if ( dynamic_cast < ::M1::A3 * > ( artifact_ ) )
   dynamic_cast < ::M1::A3 * > ( artifact_ )
    -> notify_connect_a3_sig_out ( supplier_ -> receiveSig_supplier() );
  else
   throw CORBA::NO_IMPLEMENT();
  // erzeuge Instanz der Servant-Klasse für nicht-operationalen Teil bei der
  // Nutzung von i
  ( ( OComposition* ) this ) -> _use_use = new ::M1::i__Use__::i__UseServant ( this );

  char uuid_ [ 2048 ];
  // löse Event-Kanal für sendSig auf (im use-Fall zu empfangendes Signal)
  CosEventChannelAdmin::EventChannel_var channel_
   = ::Container::Container::instance()->find_channel ( params, "sendSig" );
  if ( CORBA::is_nil ( channel_ ) )
  {
   // falls nicht erfolgreich, instanziiere Event-Kanal für receiveSig und
   // ergänze Parameter um Kanalidentifizierung
   char * oct_ = &uuid_ [ 0 ];
```

```
      Container::Container::instance()->acquire_uuid ( oct_, 2048 );
      channel_ = Container::Container::instance()->acquire_channel ( uuid_ );
      CORBA::ValueBase * base_
        = Container::ChannelIdentificationFactoryImpl::instance()->create
          ( uuid_, "sendSig", channel_ );
      ComponentModel::Container::ChannelIdentification* channel_id_
        = ComponentModel::Container::ChannelIdentification::_downcast ( base_ );
      params.length ( params.length() + 1 );
      params [ params.length() - 1 ] = channel_id_;
    }
    try
    {
      // verbinde Servant-Klasseninstanz für Interfacenutzung mit dem Event-Kanal
      CosEventChannelAdmin::ProxyPushSupplier_var proxy_push_supplier_
        = channel_->for_consumers()->obtain_push_supplier();
      proxy_push_supplier_->connect_push_consumer ( this -> _use_use->_this() );
    }
    catch ( CORBA::Exception& )
    {
      // Fehlerbehandlung
    }
    return _use_use -> _this();
  }
}
```

Die beschriebenen Mechanismen, die bei der Erzeugung von CO-Typ- und Interfacetyprepräsentation eingesetzt wurden, realisieren die Korrelation von Port-, Interaktions- und Artefaktmanagement. Mit diesen Mechanismen ist die Realisierung der Interfacenavigation sowie die Anbindung von CO-Typ-Repräsentationen an $CORE_{WARE}$ vollständig gelungen.

2.3.4.4 Zustandsattribut

Der Konzeptraum von *CORE* erlaubt die Definition von Zustandsattributen im Kontext von CO-Typen. Realisierungen des Verhaltens von CO-Typen müssen in vielen Anwendungsfällen Zugriff auf die Belegung von Zustandsattributen konkreter COs erhalten. Dementsprechend muß es Implementierungen von Artefaktklassen ermöglicht werden, auf durch $CORE_{MAP}$ produzierte Repräsentationen der Zustandsattribute zugreifen zu können. Diese Repräsentationen werden durch erzeugte C++-Klassen realisiert. Bei der Definition der Ableitungsregeln wurde auf die Integrierbarkeit mit dem in [OMG PSS2.0] definierten *Persistent-State*-Service als technologische Abstraktion persistenter Speichertechnologien Wert gelegt. Die Steuerung des potentiell parallelen Zugriffs auf die Zustandsinformationen der Repräsentation eines CO-Typs wird durch die Definition entsprechender Basisklassen in $CORE_{WARE}$ realisiert.

(Regel 69) Für alle Definitionen von Zustandsattributen im Kontext eines CO-Typs wird eine Klasse mit dem Namen **`<CO-Typ>State`** produziert. Diese Klasse spezialisiert die durch $CORE_{WARE}$ bereitgestellte Klasse **`::ComponentModel::State`**. Für jedes Zustandsattribut wird in der erzeugten Klasse eine Struktur mit dem Namen **`<Name des Zustandsattributs>StateType`** erzeugt, alle Elemente des Datentyps des Zustandsattributes werden durch Anwendung von [OMG C++Map] zu Elementen dieser erzeugten Struktur. In der erzeugten Klasse wird für alle so produzierten Strukturdefinitionen eine *Member*-Variable mit dem Namen **`_<Name des Zustansattributs>`** produziert. Im *Public*-Bereich der erzeugten Klasse wird für jede dieser Strukturdefinitionen eine Zugriffsmethode mit der Signatur **`<Name der Struktur> * <Name des Zustandsattributes> ()`** erzeugt.

Der Zugriff auf Instanzen der Strukturen, die entsprechend *Regel 69* für Zustandsattribute eines CO-Typs erzeugt wurden, ist derart gestaltet, daß parallele Rufe der Zugriffsmethoden synchronisiert erfolgen. Die Spezialisierung der Basisklasse **`::ComponentModel::State`** ist zur Integration mit Realisierungen von [OMG PSS 2.0] (z.B. durch Integration von Datenbanksystemen in $CORE_{WARE}$) vorgesehen.

(Beispiel 55) Sei die Situation aus *Beispiel 43* um Zustandsattribute des CO-Typs **O** erweitert. **O** besitze ein Zustandsattribut **o_state** vom Typ **State**. Der Datentyp dieses Zustandsattributes ist eine Strukturdefinition **State** mit den Elementen **o_state** (Enumeration) und **an_object** (Objektreferenz). Diese Situation ist in Abb. 27 dargestellt.

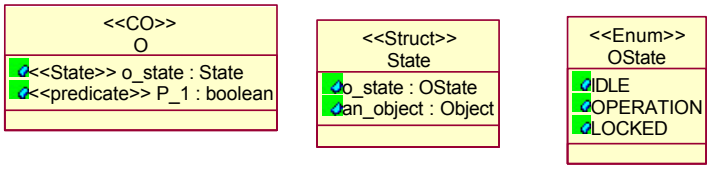


Abb. 27 Zustandsinformationen eines CO-Typs

Entsprechend *Regel 69* wird die folgende C++-Klasse erzeugt:

```
namespace M1 {
 class OState: public virtual ::ComponentModel::State
 {
  struct o_stateStateType {
    ::M1::OState o_state;
   CORBA::Object_var  an_object;
  } _o_state;
 public :
  o_stateStateType * o_state();
 };
}
```

Neben den konkreten Zustandsattributen, die im Kontext eines Entwurfsmodells explizit für einen CO-Typ definiert sind, werden in der Klasse weitere, implizite (d.h. nicht im Entwurfsmodell sichtbare) Zustandsinformationen hinterlegt. Zu diesen Informationen gehört die Menge der *Supplier*-Klasseninstanzen, die zum Management von Signalproduzenten erzeugt wurden (vgl. *Regel 64*).

(Regel 70) Die für Zustandsattribute eines CO-Typs generierte Klasse wird um *Member*-Variablen des Typs Zeiger auf die für den CO-Typ entsprechend *Regel 64* produzierten *Supplier*-Klassen sowie entsprechende Zugriffsmethoden für diese Variablen erweitert.

Durch *Regel 67* wird der Mechanismus der Konfiguration von Signal- und *Continuous-Media*-Interaktionskanälen in $CORE_{WARE}$ festgelegt. Nach der Instanziierung von *Supplier*-Klassen entsprechend *Regel 67* werden diese in den durch *Regel 70* definierten *Member*-Variablen hinterlegt und damit der Implementierung von Artefaktklassen zugänglich gemacht.

(Beispiel 56) Die produzierte Klasse **M1::OState** aus *Beispiel 55* wird entsprechend *Regel 70* um implizite Zustandsattribute für *Supplier*-Klasseninstanzen erweitert:

```
namespace M1 {
 class OState: public virtual ::ComponentModel::State
 {
  // ...
  ::M1::i__Supply__::i__Supplier_Impl* _serve;
  ::M1::i__Use__::i__Supplier_Impl* _use;
 public :
  ::M1::i__Supply__::i__Supplier_Impl* serve();
  void serve ( ::M1::i__Supply__::i__Supplier_Impl* );
  ::M1::i__Use__::i__Supplier_Impl* use();
  void use ( ::M1::i__Use__::i__Supplier_Impl* );
  // ...
 };
}
```

Der Zugriff auf die Zustandsinformationen eines COs, also auf die konkreten Instanzen der entsprechend *Regel 69* und *Regel 70* produzierten, diese Zustandsinformationen repräsentierenden Klassen, wird im Kontext der CO-Typ-*Composition*-Klasse definiert. Zu diesem Zweck wird die Definition dieser *Composition*-Klasse erweitert.

(Regel 71) Die entsprechend *Regel 44* erzeugte CO-Typ-*Composition*-Klasse wird um eine Klassenmethode **`state`** erweitert, deren Rückgabetyp Zeiger auf die entsprechend *Regel 69* und *Regel 70* produzierte Klasse **`<CO-Typ>State`** ist. Die Implementierung dieser Klassenmethode wendet die dynamische Reinterpretation von Klassen (**`dynamic_cast`**) auf die Zustandsattribute des COs an.

Bereits durch *Regel 69* wurde definiert, daß für konkrete Zustandsattribute eines CO-Typs C++-Klassen erzeugt werden, die die Klasse **`ComponentModel::State`** spezialisieren. Diese Basisklasse wird innerhalb der Klasse **`ComponentModel::ComponentBaseImpl`** zum Management von Zustandsinformationen verwendet. Nun müssen Spezialisierungen der Klasse **`ComponentModel::ComponentBaseImpl`** nur noch dynamische Zeigerumwandlung in C++ anwenden, um die Zustandsinformationen typsicher zugreifbar zu gestalten. Die diesbezügliche Realisierung von **`ComponentModel::ComponentBaseImpl`** ist gegeben durch folgendes Codefragment:

```
namespace ComponentModel {
  class State : public Monitor {};
  class ComponentBaseImpl
  {
   State * _state;
  public :
   virtual State * state_base () { return _state; }
   virtual void state_base ( State * state ) { _state = state; }
  };
}
```

(Beispiel 57) Die CO-Typ-*Composition*-Klasse für **`O`** aus *Regel 43* wird entsprechend *Regel 71* um die Methode **`state`** und deren Implementierung erweitert:

```
namespace M1 {
 class OComposition
  : public ::M1::iComposition
 {
 public :
  OState * state() {
    ComponentModel::ComponentBaseImpl * this_
     = dynamic_cast < ComponentModel::ComponentBaseImpl * > ( this );
    if ( ! this_ -> state_base() )
     this_ -> state_base ( new OState );
    return dynamic_cast < OState * > ( this_ -> state_base() );
  }
 };
}
```

Mit der Definition der Ableitungsregeln für Zustandsattribute eines CO-Typs im Entwurfsmodell wurde ein allgemeiner Mechanismus zur Repräsentation der Zustandsinformationen in der CO-Typ-Repräsentation definiert. Konkretisierungen dieses Mechanismus können z.B. Implementierungen des *Persistent-State*-Service nutzen, um Zustandsinformationen in Datenbanksystemen zu hinterlegen. Im Kontext dieser Arbeit werden derartige Konkretisierungen nicht untersucht, sie sind Bestandteil weiterführender Studien [CCM P 00].

2.3.4.5 CO-Fabriken

Die Mechanismen zur Erzeugung von COs während der Ausführung von Softwarekomponenten werden ausschließlich durch Ableitungsregeln von $CoRE_{MAP}$ erzeugt. Im Gegensatz zum CORBA-Komponentenmodell wird das Instanziierungsmanagement nicht im Entwurfsmodell erfaßt, es ist Aufgabe von $CoRE_{WARE}$ und durch $CoRE_{MAP}$ produzierten Codemodulen. *Regel 24* definiert zu diesem Zweck die Erzeugung von

CORBA-IDL-Interfacedefinitionen, die durch *Servant*-Klassen entsprechend der folgenden Regel implementiert werden.

(Regel 72) Für jede Instanz des Konzeptes CO-Typ im Entwurfsmodell wird entsprechend *Regel 24* eine CORBA-IDL-Interfacedefinition **<CO-Typ>COFactory** erzeugt. Für diese CORBA-IDL-Interfacedefinition wird eine korrespondierende *Servant*-Klasse produziert, die die CORBA-IDL-Interfacedefinition realisiert. Dabei werden Rufe der Operation **generic_create**, deren Definition in der CORBA-IDL-Basisinterfacedefinition **ComponentModel::COFactoryBase** enthalten ist, durch Delegation auf die CO-Typ-spezifische Operation **create** realisiert. Die Implementierung der Operation **create** erfolgt durch Instanziierung der für den CO-Typ entsprechend *Regel 60* erzeugten C++-Klasse **CO_<CO-Typ>**. Die Klassenmethoden, die die Realisierung der CORBA-IDL-Operationen von **ComponentModel::COFactoryBase** bereitstellen, implementieren diese Operationen durch Delegierung an die Klasse **COFactoryBaseImpl**, die durch *CORE*$_{WARE}$ definiert und realisiert wird.

(Beispiel 58) Für die in *Beispiel 43* dargestellte Situation wird für den CO-Typ **O** die folgende *Servant*-Klasse erzeugt und die Implementierung ihrer Klassenmethoden geliefert:

```
namespace M1 {
  class OCOFactoryServant
  : public ComponentModel::COFactoryBaseImpl,
    public virtual POA_M1::OCOFactory
  {
  public :
    virtual char * get_co_type ()
      { return CORBA::string_dup ( "::M1::O" ); }
    virtual ::M1::O_ptr  create ()
      {
      CO_O * retval_ = new CO_O();
      this->add_created_co ( retval_ );
      retval_->main();
      return retval_->co();
      }
    virtual ComponentModel::ComponentBase_ptr  generic_create ()
      { return this->create (); }
    virtual ~OCOFactoryServant();
    virtual ComponentModel::ComponentBase_ptr resolve_co
      (ComponentModel::ComponentKey* key)
      { return COFactoryBaseImpl::resolve_co ( key ); }
    virtual ComponentModel::ComponentKeySeq* list_cos()
      { return COFactoryBaseImpl::list_cos(); }
    virtual char * get_host ()
      { return COFactoryBaseImpl::get_host(); }
    virtual char * get_id ()
      { return COFactoryBaseImpl::get_id(); }
    virtual void set_id ( const char * )
      { COFactoryBaseImpl::set_id ( id ); }
  };
}
```

CO-Fabriken werden mittels des Registrierungsmechanismus **COFactoryRegistry** von *CORE*$_{WARE}$ verwaltet. Dieser gestattet die Registrierung sowie das Auffinden von CO-Fabriken. Der Mechanismus ist mittels CORBA-IDL folgendermaßen definiert:

```
module ComponentModel {
  module Container {
    interface COFactoryRegistry {
      void register_co_factory
        ( in ComponentModel::CoFactoryBase _factory );
      ComponentModel::CoFactoryBase resolve_co_factory
        ( in string co_type );
    };
  };
};
```

Unterschiedliche Realisierungen sind unter Zuhilfenahme von CORBA-*Object*-Services, wie z.B. *Naming*-Service oder *Trading*-Service, möglich. Die *Component-Support*-Plattform von $CoRE_{WARE}$ realisiert diesen Registrierungsmechanismus unter Verwendung des *Naming*-Service, wie im nachfolgenden Abschnitt diskutiert wird.

2.3.5 Ausführungsumgebung $CoRE_{WARE}$

Mit Hilfe der Ableitungsregeln von $CoRE_{MAP}$ ist die Repräsentation eines CO-Typs in $CoRE_{WARE}$ vollständig gelungen. Es wurden für das *Port*-, Interaktions- und Artefaktmanagement Klassen der Implementierungssprache C++ erzeugt, die die Anbindung der Realisierung der Implementierungselemente (*Business Logic*) an die CORBA-basierte *Component-Support*-Plattform von $CoRE_{WARE}$ gewährleisten. Instanzen dieser Klassen benötigen zur Ausführungszeit die Unterstützung von $CoRE_{WARE}$ zur Realisierung der folgenden Aufgaben:

- Instanziierung von CO-*Factory*-Klassen und Registrierung der zugeordneten CORBA-Objektreferenzen, um das ortsunabhängige Auffinden von CO-*Factory*-Objekten und damit die Instanziierung von COs zu gestatten,
- Instanziierung und Registrierung von Artefakt-*Factory*-Klassen zur Korrelation von Interaktions- und Artefaktmanagement,
- Bereitstellung von *Factory*-Instanzen für Repräsentationen von Signaltypen und Signalparametern sowie
- Initialisierung und Steuerung der der *Component-Support*-Plattform von $CoRE_{WARE}$ zugrundeliegenden CORBA-Infrastruktur.

Zur Realisierung dieser Aufgaben werden durch $CoRE_{WARE}$ die Mechanismen *Container* und *Component Base* definiert. Der Mechanismus *Component Base* widerspiegelt dabei die Sicht der *Component-Support*-Plattform auf konkrete CO-Typen, der Mechanismus *Container* stellt den Ausführungskontext für Repräsentationen eines oder mehrerer COs dar. Beide Mechanismen sind in der Programmiersprache C++ realisiert, für ihre externen Interaktionspunkte wurden CORBA-IDL-Definitionen bereitgestellt. Die Realisierungen beider Mechanismen sind in Form von *Software*-Bibliotheken verfügbar (*Container*-Bibliothek und *Component-Base*-Bibliothek, [BK01a]).

Jeder Verweis auf eine CO-Typrepräsentation in $CoRE_{WARE}$ ist immer ein Verweis auf **ComponentModel::ComponentBase**.

Der Zugang zum *Container*-Mechanismus ist mittels des Entwurfsmusters *Singleton* [GHJ+ 99] realisiert. Dabei wird eine Instanz der Klasse **Container** durch eine statische Methode **init(int, char**)** derselben erzeugt, und bei jedem Aufruf der statischen Klassenmethode **instance()** zurückgeliefert.

```
namespace Container {
 class Container {
  // _instance wird initial mit 0 belegt, durch Ruf von init(...) erhält
  // _instance gültigen Wert
  static Container * _instance;
  // ...
 public:
  // zur Initialisierung
  static init ( int, char ** );
  // Zugriff auf Container-Instanz
  static Container * instance();
  };
 }
```

2.3.5.1 Registrierung und Auffinden von CO-Fabriken

Durch *Regel 72* ist ein Mechanismus von **Container** mit der CORBA-IDL-Interfacedefinition **COFactoryRegistry** zur Registrierung und zum Auffinden konkreter **COFactory**-Objekte definiert. Dieser Mechanismus wird durch **Container** implementiert. Diese Implementierung nutzt den CORBA-*Naming*-

Service [OMG CORBANaS] zur Realisierung des entfernten Auffindens von CORBA-Objekten, die die CORBA-IDL-Interfacedefinition **ComponentModel::COFactoryBase** unterstützen. Dabei werden alle Bindungen einer derartigen CORBA-Objektreferenz an einen Namen des *Naming*-Service im Namenskontext (*Naming Context*) "**ComponentFactories/<Scoped-Name des CO-Typs>**" vorgenommen. Der Name, unter dem ein CORBA-Objekt vom Typ **ComponentModel::COFactoryBase** registriert wird, ist eine von der **Container**-Realisierung generierte UUID-Identifikation. Dieser Mechanismus gestattet die Registrierung mehrerer CORBA-Objekte unter verschiedenen Namen für ein und denselben CO-Typ (vgl. Abb. 28). Das Auffinden von CO-*Factory*-Objekten erfolgt mit Hilfe vollständigen Namens des CO-Typs (*Scoped Name*) durch Anfrage an den *Naming*-Service für diesen Namen. Falls für einen CO-Typ bereits entsprechende CO-Fabriken erzeugt und registriert wurden, ist der Name als **CosNaming::NamingContext**-Objekt im *Naming*-Service identifizierbar, alle Einträge können mittels der Operation **list** an diesem Objekt abgerufen werden, wobei jeder Eintrag auf ein *Factory*-Objekt für diesen Typ verweist.

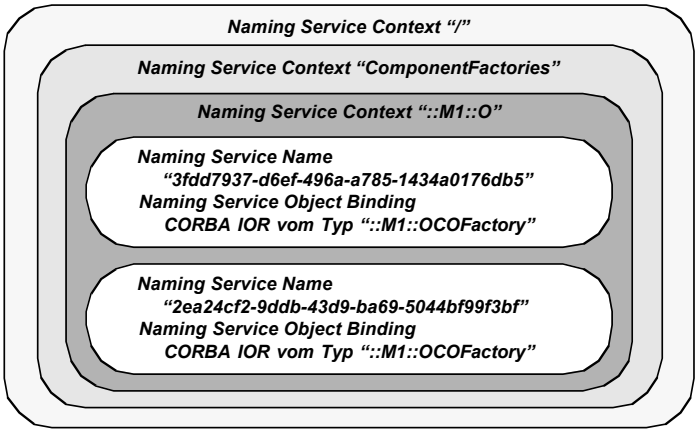


Abb. 28 Nutzung des CORBA-Naming-Service zum Registrieren und Auffinden von CO-Factory-Objekten

Der Registrierungsmechanismus für CO-Fabriken ist innerhalb des *Container*-Mechanismus durch die C++-Klasse **::Container::COFactoryRegistry** implementiert, die die CORBA-IDL-Operationen der CORBA-IDL-Interfacedefinition **COFactoryRegistry** realisiert. Diese Interfacedefinition ist wie folgt vorgenommen:

```
module ComponentModel {
  module Container {
    interface COFactoryRegistry
    {
      void register_co_factory ( in ComponentModel::CoFactoryBase _factory );
      ComponentModel::CoFactoryBase resolve_co_factory ( in string co_type );
      ComponentModel::CoFactoryBaseSeq resolve_co_factories ( in string co_type );
      ComponentModel::CoFactoryBaseSeq resolve_all_co_factories ();
    };
  };
}
```

Der Zugriff auf eine Instanz der Klasse **::Container::COFactoryRegistry** erfolgt über die *Singleton*-Instanz des Containers entsprechend dem folgenden C++-Code-Fragment:

```
::Container::Container::instance()->get_co_factory_registry()
```

2.3.5.2 Factory-Objekte für Signaltypen und Signalparameter

Gemäß den Ableitungsregeln für Signaltypen in Entwurfsmodellen (*Regel 6* und *Regel 8*) werden für diese Konstrukte CORBA-IDL-**valuetype**-Definitionen mit einer **factory**-Operation erzeugt. Ebenso werden für Typen von Signalparametern CORBA-IDL-**valuetype**-Definitionen erzeugt, die ebenfalls **factory**-Operationen enthalten können, soweit dies in einem Entwurfsmodell definiert ist. Durch Ableitungsregeln werden C++-Klassen als Implementierung dieser **valuetype**-Definitionen erzeugt, die resultierenden *Factory*-Klassen erlauben den mittels des Entwurfsmusters *Singleton* realisierten Zugriff. So kann auf die gemäß *Beispiel 28* definierte **valuetype**-*Factory*-Instanz für den Signaltyp ***Sig*** mittels des folgenden C++-Codefragmentes zugegriffen werden:

```
M1::V_var v_ = M1::VFactory::instance()->create ();
M1::Sig_var signal_ = M1::SigFactory::instance()->create_Sig ( v_ );
```

Die Instanzen aller so erzeugten **valuetype**-*Factory*-Klassen werden durch die - ebenfalls generierten - Funktionen **`void init_signal_factories ()`** und **`void init_valuetype_factories ()`** initialisiert und an der dem *Container* zugrundeliegenden ORB-Instanz registriert. Dieser Mechanismus erlaubt zum einen den einfachen Zugriff auf diese *Factory*-Objekte. Zum anderen ist sichergestellt, daß für alle in einem Entwurfsmodell verwendeten Signaltypen und Signalparameter adäquate **valuetype**-*Factory*-Instanzen zum Empfang der resultierenden **valuetype**-Instanzen (d.h. zum Empfang eines Signals und dessen Signalparametern) bereitgestellt werden.

2.3.5.3 Instanziierung und Registrierung von Artefaktfabriken

Zur Korrelation von Artefakt- und Interaktionsmanagement verwendet eine CO-Typ-*Composition*-Klasse das Konzept von Artefaktfabriken. Dabei werden für Instanzen des Konzepts Artefakt im Entwurfsmodell durch Ableitungsregeln von $CORE_{MAP}$ Artefaktfabriken erzeugt, die die im Entwurfsmodell spezifizierten Instanziierungsregeln dieser Artefakte realisieren. Instanzen dieser Artefaktfabriken werden mittels des Konzeptes *Artifact Factory Registry* im Kontext eines Containers registriert. Dieses Konzept wird durch die C++-Klasse **`ArtifactFactoryRegistry`** implementiert, die die Methoden **`register_artifact_factory`** zur Registrierung und **`resolve_artifact_factory`** zur Auflösung von konkreten **`ArtifactFactory`**-Objekten realisiert. Der Zugriff erfolgt mit **`::Container::Container::instance()->get_artifact_registry()`**. Für alle Artefaktrepräsentationen und alle CO-Typen werden im Kontext einer produzierten Softwarekomponente deren korrespondierende **`ArtifactFactory`**- bzw. CO-*Factory*-Klassen in der durch $CORE_{MAP}$ erzeugten Funktion **`void init_co_factories ()`** instanziiert. Diese Funktion ist für *Beispiel 28* folgendermaßen definiert:

```
void init_co_factories ()
{
  const ::M1::A3Factory * M1A3 = new ::M1::A3Factory;
  (::Container::Container::instance()->get_artifact_registry())
    .register_artifact_factory ( M1A3);
  const ::M1::A2Factory * M1A2 = new ::M1::A2Factory;
  (::Container::Container::instance()->get_artifact_registry())
    .register_artifact_factory ( M1A2);
  const ::M1::A1Factory * M1A1 = new ::M1::A1Factory;
  (::Container::Container::instance()->get_artifact_registry())
    .register_artifact_factory ( M1A1);
  ::M1::OCOFactoryServant * M1O = new ::M1::OCOFactoryServant;
  (::Container::Container::instance()->get_co_factory_registry())
    .register_co_factory ( M1O -> _this() );
}
```

2.3.5.4 Initialisierung und Steuerung der CORBA-Infrastruktur

Ein weiterer wesentlicher Aspekt des *Container*-Mechanismus ist das Management der CORBA-Infrastruktur, die der *Component-Support*-Plattform von $CORE_{WARE}$ zugrundeliegt. Dabei implementiert die Klasse **`Container`** zum einen die Initialisierung und Steuerung der lokal verwendeten ORB- und *Portable-Object-*

Adapter-Instanzen. Zum anderen werden mit Hilfe der generierten Funktionen **`init_co_factories`**, **`init_signal_factories`** und **`init_valuetype_factories`** die in diesen Funktionen erzeugten Instanzen der Signaltyp-/*Value*-Typ-Fabriken an der ORB-Instanz registriert sowie die CO-Fabriken aller im Kontext einer Softwarekomponente relevanten CO-Typen instanziiert und mittels des **`COFactoryRegistry`**-Mechanismus registriert.

2.3.6 Diskussion

Die Regeln zur Abbildung von Definitionen der Implementierungssicht eines Entwurfsmodells in $CoRE_{MAP}$ realisieren die Kopplung der vom Entwickler bereitzustellenden Realisierung der Repräsentationen von Implementierungselementen in Artefaktklassen an die *Component-Support*-Plattform von $CoRE_{WARE}$.

INTERFACEKONTEXT. Der gemeinsame Kontext von Elementen verschiedener Interaktionsarten an Interfacetypen ist in Form der Interface-*Composition*-Klassen auch in der programmiersprachlichen Repräsentation erhalten geblieben. Das Konzept dieses gemeinsamen Kontextes, und damit des einheitlichen Managements der Lebenszeit und des Zugriffs auf die Repräsentation eines Interfacetyps im Entwurfsmodell wird durch das Management von Instanzen der Interface-*Composition*-Klassen gelöst. Diese Klassen realisieren das Interaktionsmanagement im Kontext einer CO-Typrepräsentation.

CO-TYPKONTEXT. Die CO-Typ-*Composition*-Klassen stellen den gemeinsamen Kontext der an *Port*-Definitionen bereitgestellten bzw. genutzten Interfacetypen her. Sie bilden den Kern des *Port*- und Artefaktmanagements. Wiederum wird das Management der Lebenszeit, des Zugriffs auf angebotene bzw. genutzte Interfaces sowie der Korrelation zwischen Interaktions- und Artefaktmanagements durch das Management von Instanzen dieser Klassen gelöst. Mittels Spezialisierung der Interface-*Composition*-Klassen für angebotene bzw. genutzte Interfaces wird ein CO programmiertechnisch durch eine Instanz dieser CO-Typ-*Composition*-Klasse repräsentiert. Diese Instanz ist verantwortlich für das *Port*-, Interaktions- und Artefaktmanagement.

CO-INSTANZIIERUNG. Die Instanziierung von CO-Typ-*Composition*-Klassen - also von COs selbst - wird durch Anwendung des Entwurfsmusters "*Generic Factory*" realisiert. Für jeden CO-Typ im Entwurfsmodell wird eine CORBA-IDL-Interfacedefinition für eine derartige Fabrik produziert, die durch eine C++-Klasse implementiert wird. Die Produktion dieser Fabriken wird in ausschließlicher Verantwortung von $CoRE_{MAP}$ vorgenommen, ihre Instanziierung erfolgt durch $CoRE_{WARE}$. In Entwurfsmodellen kann beschrieben werden, daß beispielsweise ein CO ein anderes CO erzeugt. Jedoch besteht die Umsetzung dieser Entwurfsmodelldefinitionen in der Anwendung konkreter Mechanismen von $CoRE_{WARE}$.

ARTEFAKTREPRÄSENTATION. Artefakte werden durch Artefaktklassen repräsentiert, die zu Implementierungselementen im Entwurfsmodell korrespondierende Klassenmethoden enthalten. Die in einem Entwurfsmodell definierten Instanziierungsmuster werden durch die Implementierung von Artefakt-*Factory*-Klassen realisiert. Für die in *CoRE* eingeführten Instanziierungsmuster werden diese Implementierungen durch Ableitungsregeln von $CoRE_{MAP}$ erzeugt, während für die Spezifikation des Instanziierungsmusters ***`USER_DEFINED`*** die Implementierung durch den Entwickler vorgenommen wird. Die Festlegung geeigneter Instanziierungsmuster für Artefakte in Entwurfsmodellen zählt - neben der Strukturierung der Implementierung von Interaktionselementen durch Artefakte - zu den wichtigsten Entwurfsentscheidungen eines Entwicklers im Kontext der Implementierungssicht. Insbesondere ergeben sich bei der Strukturierung der Implementierung eines CO-Typs sehr viele, unterschiedliche Kombinationen zwischen Interaktionselementen von Interfacetypen und Artefakten. Hier liegt einer der grundsätzlichen Vorzüge von *CoRE*, begründet in der Vollständigkeit der Ableitungsregeln von $CoRE_{MAP}$. Durch die Ableitungsregeln ist sichergestellt, daß eine Analyse der der Codegenerierung entstammenden Softwarekomponenten auch ohne Hinzufügen der spezifischen Realisierungen von Implementierungselementen möglich ist. Beispielsweise kann die initiale Realisierung des Empfangs eines Signals in einer empirisch ermittelten Wartezeit bei der Bearbeitung des

Signals bestehen. So können Leistungsbewertungen und Integration der Softwarekomponenten mit anderen, auch auf *CORE* basierenden Softwarekomponenten bereits erfolgen, ohne daß eine konkrete Implementierung der Artefaktklassen vorliegen muß.

ANBINDUNG AN CORBA. Der gemeinsame Kontext eines Interfacetyps im Entwurfsmodell für unterschiedliche Interaktionsarten resultiert technologisch bedingt in nach Interaktionsarten getrennten CORBA-IDL-Interfacedefinitionen für die Interaktionselemente eines Interfacetyps. Durch Ableitungsregeln werden *Servant*-Klassen für jede dieser CORBA-IDL-Interfacedefinitionen erzeugt, Instanzen dieser *Servant*-Klassen werden durch Instanzen der Interface- bzw. CO-Typ-*Composition*-Klassen wiederum zusammengefaßt. Eine Folge dieser Strukturierung ist die Notwendigkeit der Delegierung im Kontext der Implementierung der *Servant*-Klassen. Die Flexibilität der Implementierungsstrukturierung eines CO-Typs mittels Implementierungselementen von Artefakten, die Interaktionselemente von Interfacetyps realisieren, impliziert die Notwendigkeit eines weiteren Delegierungsschrittes durch die Interface- bzw. CO-Typ-*Composition*-Klassen an die Artefaktklassen. Allgemein bedeutet Delegierung eine Einschränkung der Performanz einer Implementierung. Diese Aussage trifft insbesondere dann zu, wenn bei einem Delegierungsschritt die Parameter von Methoden als Wert, und nicht per Referenz übergeben werden. Die Situation wird dann verschärft, wenn die Klassenmethoden, die die Delegierung realisieren, als virtuell deklariert sind, da dann zur Ausführungszeit eine zusätzliche *Lookup*-Prozedur in der Tabelle der virtuellen Methoden durchgeführt werden muß, der Delegierungsaufruf also nicht mittels *inline* realisiert werden kann. Die Ableitungsregeln sind so definiert, daß sie die Performanzeinschränkungen möglichst gering halten:

- Die Klassenmethoden der Interface- und CO-Typ-*Composition*-Klassen sind nicht-virtuell deklariert, sie können *Inline* im Kontext der Implementierung der *Servant*-Klassen gerufen werden,
- ebenso wurden nicht-virtuelle Klassenmethoden als Repräsentationen der Implementierungselemente im Kontext von Artefaktklassen definiert, auch diese können innerhalb der *Composition*-Klassen *Inline* gerufen werden,
- als direkte Konsequenz der Anwendung der C++-*Language-Mapping*-Regeln bei der Definition der Signatur der *Composition*- und Artefaktklassenmethoden ergibt sich, daß alle Parameter sowie der Rückgabetyp stets per Referenz übergeben werden,
- parallele Invokationen von Interaktionselementen an CORBA-Interfaces werden innerhalb des *Containers* durch nebenläufige Programmkontexte (*Threads*) behandelt, und unter Einbeziehung von Synchronisationsmechanismen durch Implementierungselemente realisiert.

OPTIMIERUNG DER ABLEITUNGSREGELN. Die Ableitungsregeln lassen eine Fülle von anwendungsspezifischen Optimierungen zu. Beispielsweise wird zur Bearbeitung eines Interaktionselements durch die Instanzen der Interface- bzw. CO-Typ-*Composition*-Klassen *grundsätzlich* mittels **`resolve_artifact`** die Artefakt-*Factory*-Klasseninstanz des korrespondierenden Artefakts aufgefordert, eine Artefaktklasseninstanz zur Bearbeitung des Interaktionselements bereitzustellen. An dieser Stelle können *Caching*-Mechanismen für Artefaktinstanzen eingesetzt werden, die - solange ein *Dirty*-Kennzeichen dieser Instanz nicht die Ungültigkeit derselben anzeigt - grundsätzlich ein und dieselbe Artefaktinstanz ohne expliziten Aufruf von **`resolve_artifact`** benutzen. Um derartige Mechanismen sinnvoll in $CORE_{MAP}$ zu integrieren, sind einerseits entsprechende Spezialisierungen des Konzeptraumes $CORE_{CEPT}$ vorzunehmen und - auf der Basis dieser Spezialisierungen - die Ableitungsregeln für die Implementierung der Interface- bzw. CO-Typ-*Composition*-Klassen zu erweitern. Diese konkrete Optimierungsmöglichkeit ist als Beispiel für die Spezialisierung von *CORE* anzusehen - Anpassungen erfolgen grundsätzlich durch Spezialisierung von $CORE_{CEPT}$ und der nachfolgenden Verfeinerung der Definitionen der Ableitungsregeln von $CORE_{MAP}$.

BEZUG ZUM CORBA-KOMPONENTENMODELL. Eine der konzeptionellen Fundierungen von *CORE* ist das CORBA-Komponentenmodell. Im Rahmen dieses Standards wurde - neben den grundsätzlichen Definitionen im Kontext des CORBA-Meta-Typs **component** - ein Ansatz zur Strukturierung der Implementierung

einer CORBA-Komponente entworfen. Diese Arbeiten resultierten in der Definition dieses Ansatzes (*Component Implementation Framework, CIF*) und der Festlegung einer Beschreibungssprache (*Component Implementation Definition Language, CIDL*) zur programmiersprachenunabhängigen Spezifikation der Strukturierung der Implementierung einer CORBA-Komponente. Diese Beschreibungssprache ist in ihren grundsätzlichen Konzepten semantisch vergleichbar mit der Implementierungssicht in einem auf *CoRE* basierenden Entwurfsmodell. Insbesondere erweitern die Konzepte der Implementierungssicht diejenigen der Beschreibungssprache CIDL.

- In der Beschreibungssprache CIDL wird das Konzept *Segment* als Abstraktion programmiersprachlicher Konstrukte eingesetzt, um die Implementierung eines angebotenen Interfaces zu beschreiben. Dabei besteht eine 1:n Relation zwischen Segmenten und Interfacetypen, so daß ein Segment grundsätzlich ein oder mehrere Interfacetypen implementiert. In *CoRE* wurde eine feinkörnigere Relation zwischen Interfacetypen und Abstraktionen der diese implementierenden programmiersprachlichen Konstrukte verwendet (1:n Relation zwischen Implementierungselement eines Artefaktes und Interaktionselement eines Interfacetypen).
- Das CORBA-Komponentenmodell und damit die Beschreibungssprache CIDL trennen Interfacetypen nach Interaktionsarten. *CoRE* faßt das Konzept Interfacetyp jedoch als gemeinsamen Interaktionskontext für potentiell unterschiedliche Interaktionsarten auf - dementsprechend stellt eine *Port*-Definition den Managementkontext für derart kombinierte Interfacetypen her. Natürlich können in einem Entwurfsmodell unterschiedliche Interfacetypen für verschiedene Interaktionsarten definiert werden - eine derartige Definition ist dann semantisch vergleichbar mit Interfacedefinitionen im CORBA-Komponentenmodell. Dann können unterschiedliche Artefakte (im Sinne von Segmenten des CORBA-Komponentenmodells) zur Implementierung dieser Interfacetypen eingesetzt werden. Auch in diesem Kontext entwickelt *CoRE* das CORBA-Komponentenmodell konsequent weiter.
- Das CORBA-Komponentenmodell integriert die Interaktionsart *Continuous-Media* nicht. Während zwischen Signal- und operationaler Interaktion im Kontext einer CORBA-Komponente unterschieden wird, ist die Einbeziehung von *Continuous-Media*-Interaktionen nur durch die Definition von operationalen Interfacetypen zum Management von Audio- und Videoströmen entsprechend [OMG AVStreams] möglich. *CoRE* läßt die Definition von *Continuous-Media*-Interaktionselementen zu. Für diese Interaktionselemente werden spezielle Ableitungsregeln angewandt, um sie durch Mechanismen von $CoRE_{WARE}$ umzusetzen. Wird in einem Entwurfsmodell auf die Definition von Interaktionselementen dieser Interaktionsart verzichtet, so entspricht das Resultat der semantischen Fundierung des CORBA-Komponentenmodells.

Der Vergleich der Konzepte der Beschreibungssprache CIDL mit den Konzepten der Implementierungssicht von *CoRE* demonstriert die Weiterentwicklung des CORBA-Komponentenmodells im Rahmen dieser Arbeit. Als Konsequenz ergibt sich, daß die Grundsätze der Ableitungsregeln von $CoRE_{MAP}$ für die Konzepte der Implementierungssicht in einer auf das CORBA-Komponentenmodell eingeschränkten Form ebenfalls die Erzeugung von Softwarekomponenten aus CIDL-Beschreibungen zuzulassen versprechen. Die Untersuchungen dieser These sind insbesondere deshalb äußerst wichtig, da für die Beschreibungssprache CIDL bisher keine *Language-Mapping*-Regeln für irgendeine Programmiersprache vorgelegt wurden, insofern die Grundsätze der hier diskutierten Ableitungsregeln den am weitesten vorangetriebenen Ansatz für die Sprache CIDL überhaupt darstellen. Diese Untersuchungen sind Bestandteil einer gemeinsamen Entwicklungsinitiative zwischen der Humboldt-Universität zu Berlin und GMD Fokus, deren Ergebnisse direkt in den Standardisierungsprozeß der *Object Management Group* im Kontext der Vollendung des CORBA-Komponentenmodells einfließen.

2.4 Abbildung der Konzepte der Deployment-Sicht

Verteilte Softwaresysteme bestehen aus einer Menge von Softwarekomponenten, die auf Maschinen bereitgestellt und installiert werden. Die Ausführung der Softwarekomponenten auf diesen Maschinen erfüllt den Systemzweck durch die Kooperation der während dieser Ausführung erzeugten Repräsentationen von COs.

Eine Softwarekomponente enthält Codemodule, deren Ausführung die Identität, das Verhalten und den Zustand von COs realisieren. Softwarekomponenten werden im Entwurfsmodell durch Instanziierung des Konzeptes Softwarekomponente spezifiziert. Wichtigste Information über Softwarekomponenten ist die Spezifikation derjenigen CO-Typen, die während der Ausführung einer Softwarekomponente instanziiert werden können, für die also die Softwarekomponente alle zur Erbringung des Verhaltens und zur Realisierung der Identität benötigten Codemodule enthält.

Die Konzepte von $CoRE_{CEPT}$ zur Beschreibung dieser Zusammenhänge sind Softwarekomponente und *Realize*-Relation zwischen Softwarekomponente und CO-Typ. Das Konzept Softwarekomponente ist die Abstraktion einer Softwarekomponente, deren physikalischer Repräsentant durch Zusammenfassung der Codemodule gebildet wird, die durch Anwendung der Ableitungsregeln von $CoRE_{MAP}$ entstehen. Konkret sind entsprechend den Resultaten der Abbildung der Struktur-, Konfigurations- und Implementierungssicht Ableitungsregeln zu formulieren, die die erzeugten Codefragmente (CORBA-IDL-Definitionen und C++-Klassen mit ihren Implementierungen) zunächst in Quelldateien zusammenfassen sowie die Bildung von ausführbaren, in Softwarekomponenten enthaltenen Codemodulen aus diesen Quelldateien definieren. Die Ableitungsregeln sind dabei so zu formulieren, daß sie für eine konkrete Menge (und zwar die durch *Realize*-Relationen zwischen Softwarekomponenten- und CO-Typdefinitionen identifizierten) von CO-Typdefinitionen in einem Entwurfsmodell Quelldateien mit allen, im Kontext der Repräsentation dieser CO-Typen relevanten Codefragmenten produzieren. Dieses Prinzip impliziert eine grundsätzliche Anforderung an die Definition der Ableitungsregeln für die Konzepte der *Deployment*-Sicht:

- *Vollständigkeit* - Alle Codefragmente, die im Kontext von CO-Typen erzeugt werden, müssen in Quelldateien enthalten sein, aus denen Softwarekomponenten erzeugbar sind, die gerade diese CO-Typen realisieren.

Diese Anforderung ist selbstverständlich mittels eines simplen Mechanismus erfüllbar, der die produzierten Codefragmente für *alle* Definitionen eines Entwurfsmodells in Quelldateien zusammenfaßt. Die Anwendung dieses Mechanismus hat aber zur Folge, das für komplexe Entwurfsmodelle sehr große Softwarekomponenten entstehen, wobei u.U. nur ein minimaler Teil der enthaltenen Codemodule tatsächlich genutzt wird. Dementsprechend wird die Anforderung der Vollständigkeit ergänzt um die Anforderung der Ausschließlichkeit:

- *Ausschließlichkeit* - Eine Softwarekomponente, die eine Menge von CO-Typen realisiert, soll ausschließlich diejenigen Codemodule enthalten, die im Kontext dieser CO-Typen benötigt werden.

Dabei ist durch $CoRE_{CEPT}$ (vgl. [CoRE II], Abschnitt 3.2.1) vorgegeben, welche Definitionen eines Entwurfsmodells im Kontext eines CO-Typs benötigt werden, nämlich:

- die Definition des CO-Typs selbst sowie alle in der Definition eines CO-Typ enthaltenen Definitionen (bestimmt durch Metamodellkonstrukt ***Contained***),
- alle an enthaltenen *Port*-Definitionen definierten Interfacetypen sowie deren enthaltene Definitionen, sowie die transitive Hülle der im Metamodell definierten Relationen dieser Definitionen.

Die konkreten Definitionen von $CoRE_{CEPT}$ unterstützen die Erfüllung beider Anforderungen. Es läßt sich mit deren Hilfe die folgende Regel zur Abbildung der Konzepte der *Deployment*-Sicht formulieren.

(Regel 73) Für alle Instanzen des Konzeptes CO-Typ im Entwurfsmodell, zu denen eine Instanz des Konzeptes Softwarekomponente eine *Realize*-Relation besitzt, werden diejenigen Entwurfsmodelldefinitionen, die durch Instanzen der im Metamodell definierten Relationen (Generalisierung und Assoziationen) transitiv erreichbar sind, in die Codegenerierung entsprechend den bisher definierten Ableitungsregeln einbezogen.

Die Überführung der Quelldateien, die entsprechend dieser Regel die produzierten Codefragmente für Entwurfsmodelldefinitionen aufnehmen, in ausführbare Softwarekomponenten erfolgt durch einen Kompilationsschritt, unterstützt durch eine *Make*-Datei. Die Aufteilung dieser Codefragmente in Quelldateien ist tatsächlich vollkommen willkürlich, eine „vernünftige", grobe Aufteilung ist sicherlich die Separierung von CORBA-IDL-Definitionen in Quelldateien mit dem Suffix „**`idl`**", C++-Klassendeklarationen in Quelldateien mit dem Suffix „**`h`**" und C++-Implementierungsdateien mit dem Suffix „**`cpp`**" sowie eine Verfeinerung dieser Aufteilung in unterschiedliche Quelldateien für *Servant*-, *Composition*- und Artefaktklassen sowie separate Quelldateien von $CoRE_{WARE}$. In der prototypischen Implementierung von Codegeneratoren für $CoRE_{MAP}$, die die hier vorgestellten Ableitungsregeln realsieren, wurde gerade dieses Schema umgesetzt.

Eine der wesentlichen Anforderungen bei der Konzeption der *Component-Support*-Plattform und der Definition der Ableitungsregeln bestand in der Realisierung des Konfigurationsmanagements durch allgemeine, in $CoRE_{WARE}$ integrierte Werkzeuge. Insbesondere muß dabei sichergestellt sein, daß derartige Konfigurationswerkzeuge unter Kenntnis der CO-Typ-Definitionen konkrete Konfigurationen für eine Menge von COs vornehmen können. Dieser Anforderung wurde durch die Definition generischer *Port*-Operationen (**provide**, **provide_supply** und **connect**) sowie generischer Instanziierungsoperationen (**generic_create**) im Kontext der CORBA-IDL-Interfacedefinitionen **::ComponentModel::ComponentBase** und **::ComponentModel::COFactoryBase** entsprochen. Mit Hilfe dieser CORBA-IDL-Interfacedefinitionen und dem Mechanismus **COFactoryRegistry** des *Containers* ist es Konfigurationswerkzeugen möglich, ohne Kenntnis der CORBA-IDL-Definitionen der Repräsentation von CO-Typen Konfigurationen (insbesondere die initiale Konfiguration) durch Instanziierung von COs und Verbinden von *Port*-Definitionen dieser COs aufzubauen. Konkret können solche Konfigurationswerkzeuge mit der ausschließlichen Kenntnis der Entwurfsmodelle von verteilten Softwaresystemen arbeiten, die mittels *CoRE* entworfen wurden. Ein solches Konfigurationswerkzeug wurde durch den Autor unter Nutzung des *Add-In*-Mechanismus des Produktes Rational Rose [RationalRose] durch Anbindung an $CoRE_{WARE}$ erstellt. Mit diesem Werkzeug ist zur Ausführungszeit die Auflösung der aktuellen Konfiguration von CO-Fabriken und durch sie erzeugten COs, die Instanziierung von COs sowie die Verbindung der *Port*-Definitionen von COs möglich. Als graphische Oberfläche dieses Werkzeugs (vgl. Abb. 29) wurde die UML-Notation für *Collaboration*-Diagramme gewählt - es werden COs dargestellt, deren CO-Typ im gleichen Entwurfsmodell definiert ist (vgl. Kapitel 5).

Das gegenwärtig realisierte Werkzeug läßt die manuelle Erstellung von Konfigurationen von COs und ihren *Port*-Definitionen zu. In vielen Umgebungen besteht der Wunsch, mittels geeigneten Werkzeugen auf der Basis von Konfigurationenbeschreibungen automatisch COs zu erzeugen und *Port*-Definitionen dieser COs zu verbinden. Dieser Automatismus ist insbesondere für die initiale Konfiguration des verteilten Softwaresystems interessant. Diese automatische Konfiguration wird wiederum durch die generischen CORBA-IDL-Interfacedefinitionen von **::ComponentModel::ComponentBase** und **::ComponentModel::COFactoryBase** unterstützt.

Konfigurationsbeschreibungen können in unterschiedlichen Notationen vorliegen, die zu verwendenden Konzepte sind durch die Definitionen des Metamodells in $CoRE_{CEPT}$ erklärt. Im Rahmen des EURESCOM Projektes P924 „*Deployment and Configuration Support for Distributed PNO Applications*" [EUP924] wurde eine derartige konkrete Notation definiert [EUP924a]. Im gleichen Kontext wurde eine Werkzeugumgebung entwickelt und mehrfach demonstriert, die Softwarekomponenten, die auf der Basis von *CoRE* entwickelt wurden, automatisch auf Maschinen verfügbar macht, installiert und initiale Konfigurationen von COs und ihren *Port*-Definitionen erzeugt.

2.5 Abbildung der Konzepte der Interaktionssicht

Eine der wesentlichen Eigenschaften von Telekommunikationssoftwaresystemen ist, daß sie i.allg. Leistungen erbringen, die *abrechenbar* sind. Die Abrechnung solcher Leistungen erfolgt auf der Basis von Güteeigen-

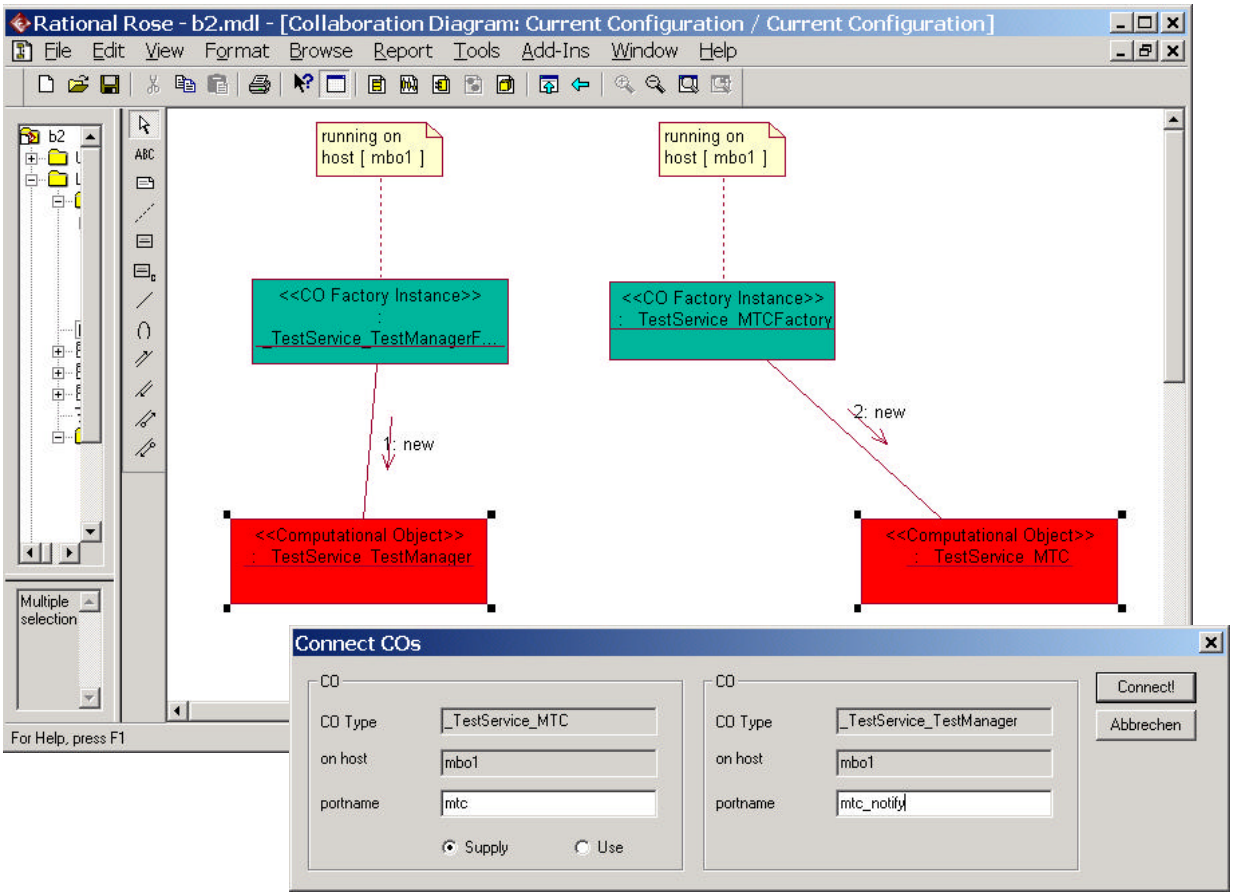


Abb. 29 Konfigurationswerkzeug der Component-Support-Plattform

schaften, die zwischen dem Nutzer der Dienstleistungen und deren Erbringer *vereinbart* und durch den Erbringer der Leistungen *garantiert* werden.

Dieser Anforderung entsprechend gestattet *CORE* die Erfassung von Güteeigenschaften in Entwurfsmodellen. Güteanforderungen werden dabei im Kontext von COs definiert - sie beschreiben *geforderte* Charakteristika der Interaktion dieser Objekte. Dabei wurden die zum Einsatz kommenden Konzepte von $CORE_{CEPT}$ gerade so formuliert, daß sie die Beschreibung von unterschiedlichen Umgebungsbedingungen zulassen, in denen Interaktionen zwischen COs auftreten können. Diese Beschreibung erfolgt fallbasiert - ein Bindungsfall wird charakterisiert durch eine Menge von Prädikaten. Die Auswertung aller Prädikate, die für CO-Typen definiert sind, deren Instanzen in Interaktion treten, identifiziert einen Bindungsfall, der im Entwurfs-

modell spezifiziert ist. Die Ableitungsregeln von $CoRE_{MAP}$ für diese Konzepte und $CoRE_{WARE}$ als Zielumgebung müssen demzufolge sicherstellen, daß

- Mechanismen realisiert werden können, mit denen die Hinterlegung von und der Zugriff auf geforderte Dienstgüteeigenschaften möglich ist; dabei sind sowohl die geforderten Dienstgüteeigenschaften von Relevanz als auch die Definition der Kontrakttypen, über denen diese Forderungen definiert sind,
- zur Ausführungszeit die Existenz von Umgebungsbedingungen auswertbar ist, die zur Identifikation von Bindungsfällen führen,
- daß anhand der Auswertung dieser Umgebungsbedingungen entweder ein Bindungsfall des Entwurfsmodells oder aber der *Default*-Bindungsfall identifiziert werden kann,
- Mechanismen in $CoRE_{WARE}$ existieren, die - basierend auf der Identifikation des Bindungsfalles und der Bestimmung der geforderten Dienstgüteeigenschaften für diesen Bindungsfall - konkrete Dienstgüteeigenschaften zwischen den COs verhandeln, die an einem Bindungsfall beteiligt sind sowie
- Mechanismen von $CoRE_{WARE}$ die Überwachung und Sicherstellung dieser verhandelten Dienstgüteeigenschaften gewährleisten.

Im folgenden werden die Mechanismen diskutiert, die zur Identifikation von Bindungsfällen führen. Entsprechend den Definitionen von $CoRE_{CEPT}$ ist dabei zunächst zu untersuchen, wie Prädikate ausgewertet werden, die für CO-Typen definiert sind.

Prädikat als Konzept in $CoRE_{CEPT}$ ist eine Abstraktionen von zu wahr oder falsch evaluierbaren Ausdrükken. In einem initialen Ansatz kann die Abbildung von Prädikaten durch eine Menge von Operationen im Kontext der CORBA-IDL-Interfacedefinition erfolgen, die entsprechend *Regel 22* für einen CO-Typ erzeugt wird. Dabei wird für jedes Prädikat, das für einen CO-Typ definiert ist, eine CORBA-IDL-Operation erzeugt. Die Auswertung der Prädikate erfolgt dann mittels einer Sequenz von Rufen dieser Operationen. Diese Abbildungsstrategie hat den Vorteil, daß zur Identifikation konkreter Bindungsfälle selektiv diejenigen Prädikate ausgewertet werden, die im Kontext eines Bindungsfalles relevant sind. Andererseits ergibt sich der Nachteil, daß zur Identifikation komplexer Bindungsfälle u.U. eine große Anzahl von Operationsrufen ausgeführt werden muß - resultierend in der Beeinträchtigung der Performanz des Gesamtsystems. Es ist also eine Abbildungsstrategie zu finden, die zum einen die selektive Auswertung von Prädikaten gestattet, auf der anderen Seite diese Auswertung mittels weniger oder eines einzigen CORBA-Operationsrufes ermöglicht.

(Regel 74) Für jede Instanz des Konzeptes Prädikat im Entwurfsmodell wird eine CORBA-IDL-**valuetype**-Definition mit dem Namen des Prädikats in der CORBA-IDL-Moduldefinition erzeugt, die zum Namensraum im Entwurfsmodell korrespondiert, in dem das Prädikat definiert wurde. Diese **valuetype**-Definition spezialisiert **::ComponentModel::Predicate** unter Nutzung des CORBA-IDL-Konzeptes **truncatable**, und definiert eine *Factory*-Operation **create_<Name des Prädikates>**.

Die CORBA-IDL-**valuetype**-Definition **::ComponentModel::Predicate** definiert dabei die Sicht der *Component-Support*-Plattform auf Prädikate, ohne den konkreten Typ eines Prädikates zu kennen:

```
module ComponentModel {
  valuetype Predicate  {
    enum PredicateValue {
      VALUE_INIT,  VALUE_NOT_APPLICABLE,
      VALUE_TRUE,  VALUE_FALSE
     };
    public PredicateValue predicate_value;
    public string predicate_name;
    PredicateValue evaluate ();
    factory create_predicate ( in string name );
  };
  typedef sequence < Predicate > SeqPredicate;
};
```

Das Prinzip der Spezialisierung unter Verwendung des CORBA-IDL-Mechanismus **truncatable** wird eingesetzt, um zu ermöglichen, daß $CORE_{WARE}$ mit Instanzen des Typs **Predicate** arbeiten kann, obwohl in Realisierungen von CO-Typen konkrete Ableitungen eingesetzt werden, die $CORE_{WARE}$ i.allg. unbekannt sind. Die Implementierung der Auswertung eines Prädikates wird durch den Entwickler durch Realisierung der Methode **evaluate** innerhalb der C++-Repräsentation von CORBA-IDL-**valuetype**-Definitionen für konkrete Prädikate vorgenommen. Auf der Grundlage dieser Abbildung wird die Definition der CORBA-IDL-Interfacedefinitionen für CO-Typen um eine Operation zur Initiierung der Auswertung von Prädikaten erweitert, indem der Basisinterfacedefinition **ComponentModel::ComponentBase** die Operation **evaluate** hinzugefügt wird:

```
module ComponentModel {
  interface ComponentBase {
    // ...
    void evaluate ( inout SeqPredicate predicates );
    };
  };
```

Wie im Falle aller weiteren Operationen der CORBA-IDL-Interfacedefinition eines CO-Typs wird entsprechend *Regel 40* die korrespondierende **Servant**-Klasse um die Methode **evaluate** als Realisierung der CORBA-IDL-Operation **evaluate** erweitert. Die Implementierung dieser Methode nutzt Delegierung an die CO-Typ-*Composition*-Klasse. Sie erzeugt für jedes Prädikat, das in der Liste der Prädikate enthalten ist, eine Instanz der Spezialisierung von **Predicate**, die durch den Prädikatsnamen identifizierbar ist. Das dementsprechend bearbeitete Prädikat wird aus der Liste der Prädikate entfernt. An dieser Instanz wird die Methode **evaluate** gerufen, die vom Entwickler bereitzustellen ist, und die die *Member*-Variable **predicate_value** mit einem konkreten Wert belegt. Die Instanz des spezialisierten Prädikats wird der Liste der Prädikate hinzugefügt. Falls im Kontext eines CO-Typs ein Prädikat aus der übergebenen Liste nicht unterstützt ist, wird der Liste der Prädikate eine Instanz von **Predicate** mit dem Wert **VALUE_NOT_APPLICABLE** der *Member*-Variablen **predicate_value** hinzugefügt.

(Regel 75) Die Implementierung der entsprechend *Regel 44* erzeugten Methode **`void evaluate( PredicateSeq& predicates, const PortableServer::ServantBase *)`** der CO-Typ-*Composition*-Klasse erzeugt eine Liste **predicates_return** und wertet alle in der Liste **predicates** der Prädikate enthaltenen Prädikate entsprechend des folgenden Verfahrens aus:

- entnehme der Liste **predicates** die erste Instanz und entferne diese aus **predicates**,
- falls der durch die **Member**-Variable **predicate_name** identifizierte Typ des Prädikates im Kontext des CO-Typs unterstützt wird, erzeuge eine Spezialisierung von **Predicate** unter Nutzung der mit dem *Container* registrierten *Value*-Fabrik **<Prädikatname>Factory** und deren Methode **create_<Prädikatname>**;
- falls der identifizierte Typ des Prädikates nicht unterstützt wird, erzeuge eine Instanz von **Predicate** und belege die *Member*-Variable **predicate_value** mit **Predicate::VALUE_NOT_APPLICABLE**;
- rufe an der erzeugten Instanz die Methode **evaluate**,
- füge die Instanz der Liste **predicates_return** hinzu,
- falls alle in **predicates** enthaltenen Instanzen ausgewertet wurden, überschreibe **predicates** mit **predicates_return**.

(Beispiel 59) Seien im Kontext von *Beispiel 33* zwei Prädikate ***P_1*** und ***P_2*** definiert. Dann werden durch *Regel 74* und *Regel 75* die folgenden CORBA-IDL- bzw. C++-Definitionen erzeugt:

```
module M1 {
  valuetype P_1 : truncatable ComponentModel::Predicate  {
    factory create_P_1 ();
  };
   valuetype P_2 : truncatable ComponentModel::Predicate {
    factory create_P_2 ();
  };
};
```

```
namespace M1 {
 class OComposition
  :  public ::M1::iComposition
 {
  // ...
  public :
   void evaluate
    ( ComponentModel::PredicateSeq& predicates, const PortableServer::ServantBase *)
   {
    ComponentModel::PredicateSeq predicates_return;
    for ( unsigned long i = 0; i < predicates.length(); i++ )
    {
     if ( strcmp ( predicates[i]->predicate_name(), "::M1::P_1" ) == 0 )
     {
      ::M1::P_1 * p_ = 1::P_1Factory::instance()->create_P_1();
      p_->evaluate();
      predicates_return.length ( predicates_return.length() + 1 );
      predicates_return [ predicates_return.length() - 1 ] = p_;
     }
     else if ( strcmp ( predicates[i]->predicate_name(), "::M1::P_2" ) == 0 )
     {
      // Auswertung für P_2
     }
     else
     {
      ComponentModel::Predicate * p_
       = ComponentModel::PredicateFactory::instance()->create_predicate
        ( predicates[i]->predicate_name() );
      p_->predicate_value = ComponentModel::Predicate::VALUE_NOT_APPLICABLE;
      predicates_return.length ( predicates_return.length() + 1 );
      predicates_return [ predicates_return.length() - 1 ] = p_;
     }
    }
   }
 };
}
```

Mit Hilfe dieser Ableitungsregel ist nun definiert, auf welche Weise innerhalb der Realisierung eines CO-Typs Prädikate ausgewertet werden. In *CORE*$_{WARE}$ können damit die im Kontext eines zu identifizierenden Bindungsfalles relevanten Prädikate durch konkrete COs evaluiert werden - ein Bindungsfall kann also unter Voraussetzung der Kenntnis der relevanten Prädikate identifiziert werden.

Die Korrelation von Bindungsfällen und in diesen relevanten Prädikaten wird durch einen *Repository*-Mechanismus von *CORE*$_{WARE}$ verwaltet. Die Beschreibung von Bindungsfällen eines Entwurfsmodells wird durch entsprechende Einträge im *Repository*-Mechanismus repräsentiert. Die Beschreibung von Bindungsfällen in Entwurfsmodellen wird in XML-Dokumente überführt. Nachfolgende werden *Repository*-Einträge aus den Elementen eines XML-Dokumentes erzeugt. Dazu wird die folgende CORBA-IDL-Interfacedefinition **ComponentModel::BindingCaseRepository::BindingCase** genutzt:

```
module ComponentModel {
 module BindingCaseRepository {
  interface BindingCase {
   struct RequiredPredicates {
    CORBA::RepositoryId co_type;
    CORBA::RepositoryId port;
    PredicateSeq predicates;
   };
   typedef sequence < RequiredPredicates > RequiredPredicatesSeq;
   readonly attribute CORBA::RepositoryId binding_case_id;
   readonly attribute RequiredPredicatesSeq required_predicates;
  };
 };
};
```

Dabei dient die Strukturdefinition **RequiredPredicates** der Korrelation von CO-Typ und *Port*-Definition mit einer Menge von Prädikaten, die im Kontext eines Bindungsfalles für die beteiligte Instanz des CO-Typs erfüllt sein müssen. Die Liste aller derartigen Korrelationen wird durch das **readonly**-Attribut **required_predicates** bereitgestellt. Die *Repository*-Identifikation des Namens des Bindungsfalles im Entwurfsmodell kann mittels des Attributs **binding_case_id** gelesen werden.

Mit Hilfe dieser Definitionen können aus einem Entwurfsmodell nun derartige *Repository*-Einträge erzeugt werden. Der *Repository*-Mechanismus selbst ist ebenfalls mittels CORBA-IDL definiert:

```
module ComponentModel {
  module BindingCaseRepository {

    typedef sequence < BindingCase > BindingCaseSeq;
    interface BindingCaseRep
    {
      exception NotFound {};
      exception AlreadyStored {};

      struct ParticipatingObject
        {
        CORBA::RepositoryId co_type;
        CORBA::RepositoryId port;
        ComponentModel::ComponentBase co_ref;
        };

      typedef sequence < ParticipatingObject > ParticipatingObjectSeq;

      BindingCaseSeq resolve_binding_case ( in ParticipatingObjectSeq objects )
        raises ( NotFound );

      BindingCase create_binding_case (
        in BindingCase::RequiredPredicatesSeq required_predicates,
        in CORBA::RepositoryId binding_case_id )
        raises ( AlreadyStored );

      void remove_binding_case ( in CORBA::RepositoryId binding_case_id )
        raises ( NotFound );
    };
  };
};
```

Der **BindingCaseRepository**-Mechanismus bietet Operationen zum Hinzufügen (**create_binding_case**), Entfernen (**remove_binding_case**) und Auffinden (**resolve_binding_case**) von Bindungsfallbeschreibungen an. Das Hinzufügen von Beschreibungen erfolgt durch ein Werkzeug, das XML-Dokumente verarbeitet, die aus Bindungsfallspezifikationen in Entwurfsmodellen gewonnen wurden. Die Operation **resolve_binding_case** liefert alle Bindungsfallbeschreibungen zurück, die für eine Menge von CO-Typen und deren *Port*-Definitionen vorliegen, diese werden beschrieben durch die Strukturdefinition **ParticipatingObject**.

Zur Identifikation eines konkreten Bindungsfalles bietet $CORE_{WARE}$ im Kontext eines *Containers* einen generischen Mechanismus, der zur Initiierung der Interaktion einer Menge von COs genutzt werden kann. Die Nutzung dieses Mechanismus kann sowohl durch die Repräsentation eines COs selbst als auch durch externe, allgemeine Konfigurationswerkzeuge erfolgen. Dieser Mechanismus identifiziert mit Hilfe von **BindingCaseRepository** für eine gegebene Menge von COs einen konkreten Bindungsfall bzw. den *Default*-Bindungsfall. Dabei werden mittels der durch *Regel 74* bzw. *Regel 75* definierten Mechanismen alle Prädikate durch Rufe von **evaluate**-Operationen an den COs überprüft, die im Kontext der Bindungsfälle relevant sind, die durch das Ergebnis von **resolve_binding_case** referenziert werden. Falls die Auswertung ergibt, daß entweder mehr als ein oder kein Bindungsfall identifiziert werden kann, resultiert die Annahme des *Default*-Bindungsfalles.

2.5.1 Diskussion

Zur Identifikation von Bindungsfällen und zur Moderation der Verhandlung von Dienstgüteeigenschaften können konkrete, auf den Definitionen von $CoRE_{CEPT}$ basierende Ableitungsregeln und Mechanismen von $CoRE_{WARE}$ definiert werden. Im Gegensatz dazu stellen Überwachung und Sicherstellung dieser Eigenschaften eine komplexe, vom Typ der konkreten Eigenschaften abhängige Problematik dar. Eine umfassende, insbesondere geschlossene Lösung dieser Problemstellung ist schwer zu erreichen. Vielmehr wird hier ein generischer Ansatz verfolgt, der die Integration konkreter Mechanismen in $CoRE_{WARE}$ zum Ziel hat, die für konkrete Dienstgüteeigenschaften nutzbar sind.

Dabei stellt ein *Repository*-Mechanismus von $CoRE_{WARE}$ die Korrelation von geforderten Dienstgüteeigenschaften und *Runtime-Policy*-Elementen her, die zur Sicherstellung dieser Eigenschaften zu nutzen sind. *Runtime-Poliy*-Elemente haben dabei einen *Policy*-Typ, unter dem Mechanismen, die diesen unterstützen, in der Ausführungsumgebung für Softwarekomponenten registriert sind. Dieser Ansatz ist in [BKvH 00] präzisiert und wird derzeit in einem Forschungsprojekt mit *KPN Research* unter Beteiligung des Autors realisiert.

2.6 Realisierung von $CoRE_{MAP}$

Die Ableitungsregeln sind durch die Codefragmente einzuhalten, die ein diese Regeln implementierendes Werkzeug produziert. Die Ableitungsregeln haben zwar konstruktiven Charakter, sie implizieren jedoch *keinen* Algorithmus (im Sinne einer sequentiellen Abfolge der Anwendung von Regeln) für ein derartiges Werkzeug. Das durch den Autor realisierte Werkzeug nutzt den folgenden Algorithmus [BK 01a]:

1. Berechnung der abzubildenden Elemente des Entwurfsmodells und deren Abhängigkeiten,
2. Berechnung der zu erzeugenden Codemodule,
3. Erzeugung einer *Make*-Datei,
4. Erzeugung der CORBA-IDL-Definitionen für die der Struktur- und Konfigurationssicht zuzuordnenden Elemente des Entwurfsmodells,
5. Erzeugung der *Servant*-, Interface- und CO-Typ-*Composition*-Klassen, ausgehend von den produzierten CORBA-IDL-Definitionen,
6. Erzeugung der Implementierungen der produzierten *Servant*-, Interface- und CO-Typ-*Composition*-Klassen,
7. Erzeugung der Artefaktklassen und deren Implementierung,
8. Erzeugung der CO-Repräsentationen als Spezialisierung der Interface- und CO-Typ-*Composition*-Klassen und deren Implementierung (Kopplung der Artefaktklassen an die CO-Typ-*Composition*-Klassen),
9. Erzeugung der Klassen für Zustandsattribute.

Im Anschluß an die Anwendung des Werkzeuges auf ein Entwurfsmodell werden die erzeugten Artefaktklassen durch den Entwickler komplettiert.

KAPITEL 3

Interaktionsabläufe

Im folgenden werden Abläufe der Interaktionen zwischen COs und innerhalb von COs präsentiert, um die Wirkungsweise der durch $CORE_{MAP}$ produzierten Softwarekomponenten und die enthaltenen Codemodule zu illustrieren. Alle Interaktionsabläufe werden mittels UML-Sequenzdiagrammen dargestellt.

3.1 Erzeugung und Registrierung von Artefaktfabriken

Während der Initialisierung eines *Containers* durch $CORE_{WARE}$ werden alle CO- und Artefakt-*Factory*-Klassen instanziiert und registriert, die in einer durch diesen *Container* ausgeführten Softwarekomponente enthalten sind. Diese Initialisierung wird durch die in Abschnitt 2.3.5 vorgestellte, durch Ableitungsregeln produzierte Funktion **`void init_co_factories()`** vorgenommen.

Im in *Beispiel 43* dargestellten Szenario werden für die im Entwurfsmodell definierten Artefakte ***A1***, ***A2*** und ***A3***, die Artefakt-*Factory*-Klassen **`A1Factory`**, **`A2Factory`** und **`A3Factory`** (vgl. *Regel 55*) sowie die *Factory*-Klasse **`OCOFactory`** für die CO-Typrepräsentation von ***O*** erzeugt. Instanzen dieser Klassen werden durch den *Container* angelegt und mittels der Methode **`register_artifact_factory`** der Klasse **`ArtifactFactoryRegistry`** bzw. **`register_co_factory`** der Klasse **`COFactoryRegistry`** registriert (vgl. *Regel 56*). Dieser Ablauf ist für **`A1Factory`** und **`OCOFactoryServant`** in Abb.30 dargestellt.

3.2 Erzeugen von CO-Repräsentationen

Das Erzeugen von COs im Kontext eines *Containers* wird durch den Registrierungsmechanismus für CO-Fabriken unterstützt. Für die in *Beispiel 43* beschriebene Situation widerspiegelt Abb. 31 das Erzeugen eines COs auf Anforderung durch die Umgebung dieses instanziierten COs.

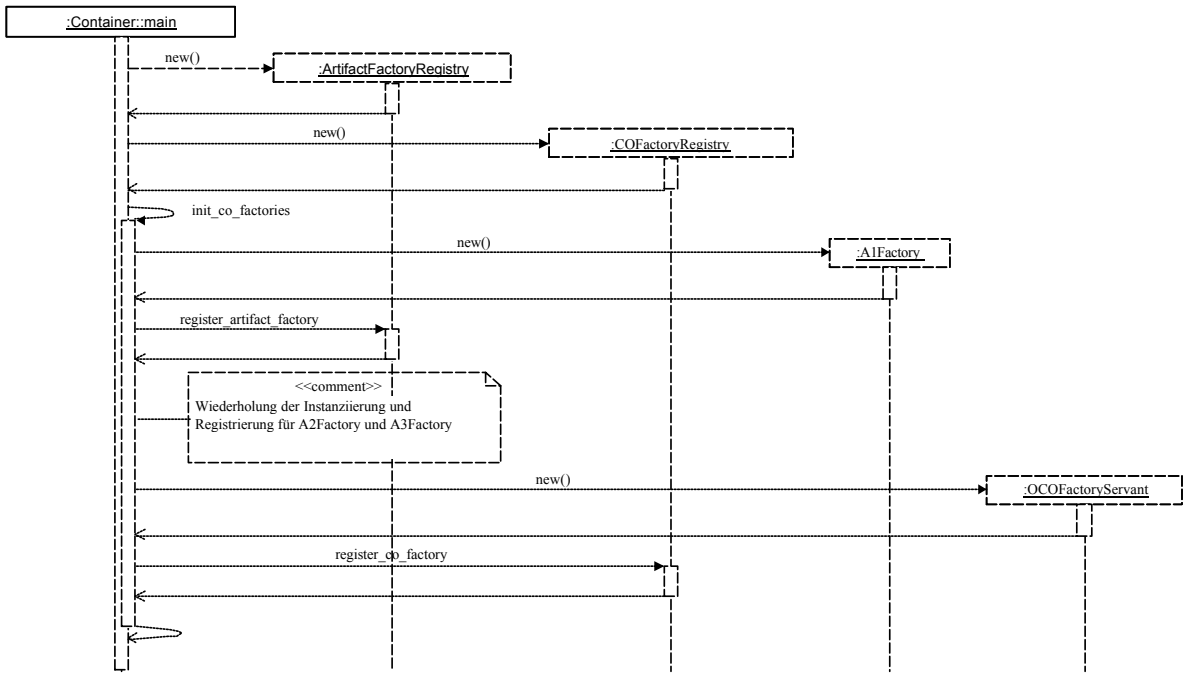


Abb. 30 Erzeugung und Registrierung von Artifact-Factory-Klasseninstanzen

Dabei wird zunächst durch die unterstützende Ausführungsumgebung (vgl. Abschnitt 2.3.5) im Konstruktor der Klasse **`Container`** eine Instanz der Klasse **`COFactoryRegistry`** erzeugt. Nachfolgend werden für alle CO-Typen, die durch einen *Container* unterstützt werden, Instanzen der entsprechend *Regel 72* produzierten Klassen **`<CO-Typname>COFactoryServant`** erzeugt und an der **`COFactoryRegistry`**-Instanz registriert. Diese Aktionen werden während der Initialisierung eines Containers ausgeführt.

Anschließend ist es der Umgebung möglich, Instanzen der durch einen *Container* unterstützten CO-Typen anzufordern. Dazu benutzt die Umgebung die durch den Container bereitgestellte Operation **`resolve_co_factory`** (vgl. Abschnitt 2.3.5) und übergibt die Identifikation des angeforderten CO-Typs. Falls dieser Typ durch den in der Anforderung adressierten *Container* unterstützt wird, erhält der Rufer eine CORBA-Interfacereferenz vom Typ **`ComponentModel::COFactoryBase`**. Mit Hilfe dieser Referenz kann die Umgebung nachfolgend an der referenzierten **`ComponentModel::COFactoryBase`**-Instanz mittels **`generic_create`** (vgl. *Regel 24*) eine CO-Repräsentation anfordern. Diese Anforderung wird durch die **`<CO-Typname>COFactoryServant`**-Instanz durch Delegierung auf die CO-Typ-spezifische **`create`**-Operation implementiert, und der Rückgabewert (CORBA-Objektreferenz vom Typ der CORBA-IDL-Interfacedefinition, die entsprechend *Regel 22* und *Regel 24* für einen CO-Typ im Entwurfsmodell produziert wurde) an den Rufer von **`generic_create`** zurückgegeben. Die **`create`**-Operation erzeugt entsprechend *Regel 44* eine Instanz der Klasse **`CO_<CO-Typname>`**, die gerade das CO repräsentiert (vgl. *Regel 72*).

3.3 Instanziierung der CO-Typrepräsentation

Die Korrelation von Port- und Interaktionsmanagement wird durch die Klasse **`CO_<CO-Typ>`** realisiert, die entsprechend *Regel 60* die erzeugte CO-Typ-*Composition*-Klasse sowie alle diejenigen Interface-*Composition*-Klassen für Interfacetypen im Entwurfsmodell spezialisiert, die entweder in *Requires*- oder *Supports*-Relation zu dem CO-Typ stehen. Während der Konstruktion der Klasse **`CO_<CO-Typ>`** werden Instanzen der entsprechenden *Servant*-Klassen angelegt und in *Member*-Variablen hinterlegt.

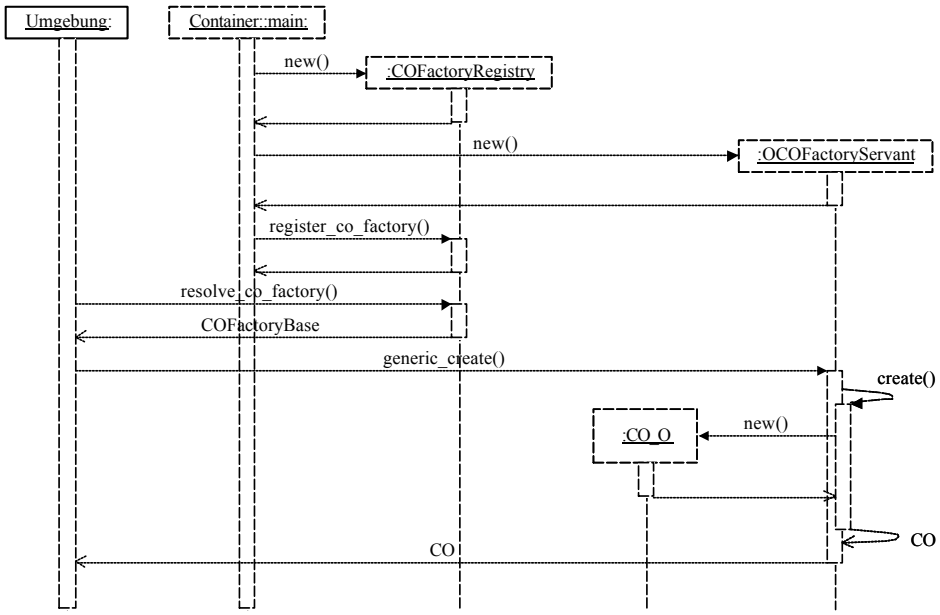


Abb. 31 Erzeugen von CO-Repräsentationen

Für die in *Beispiel 43* dargestellte Situation werden die in Abb. 32 widerspiegelten Instanziierungsschritte bei der Erzeugung einer Instanz der Klasse **`CO_O`** vorgenommen. Zunächst werden die Konstruktoren der Basisklassen **`ComponentBaseImpl`** und **`iComposition`** gerufen. Anschließend wird der Konstruktor der Klasse **`OComposition`** ausgeführt, der für die Erzeugung der *Servant*-Klasseninstanzen in Abhängigkeit von den *Port*-Definitionen im Entwurfsmodell verantwortlich ist (vgl. *Regel 45*). Der CO-Typ **`O`** im Entwurfsmodell besitzt eine *Provided-Port*-Definition, die auf einer mit **`serve`** bezeichnete *Supports*-Relation zum Interfacetyp **`i`** basiert. Dementsprechend wird eine Instanz der Klasse **`iServant`** sowie eine Instanz der Klasse

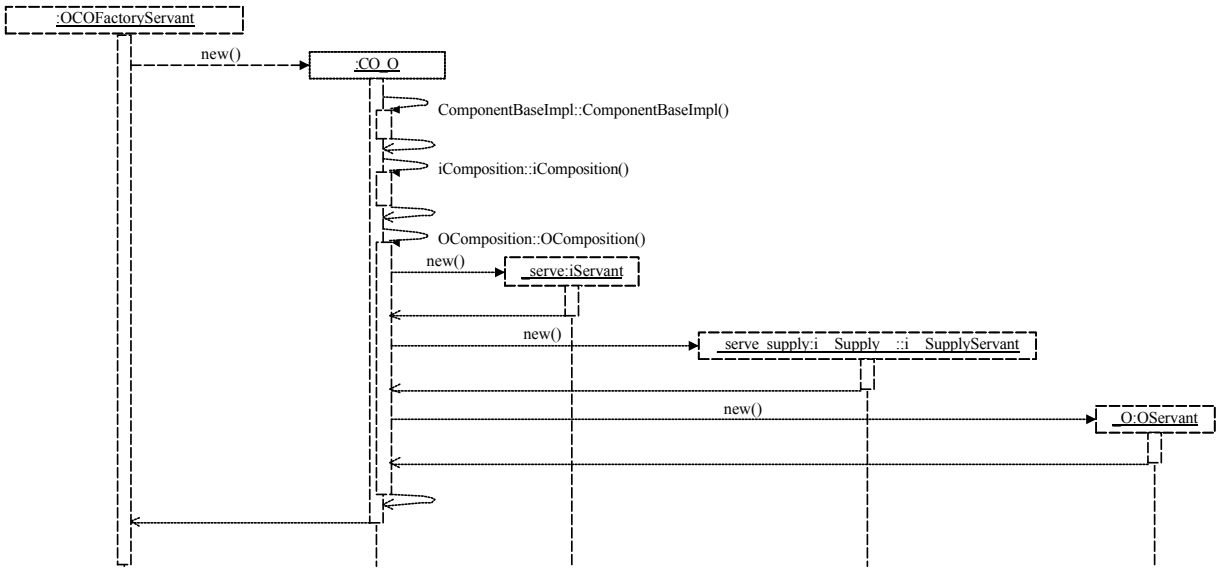


Abb. 32 Erzeugung von Composition- und Servant-Klasseninstanzen

i__Supply__::i__SupplyServant erzeugt und in den angelegten *Member*-Variablen **_serve** bzw. **_serve_supply** hinterlegt. Zur Realisierung der CO-Typfunktionalität wird eine Instanz der Klasse **OServant** erzeugt und in der produzierten *Member*-Variablen **_O** abgelegt.

3.4 Interfacenavigation

Interfacenavigation bezeichnet die Eigenschaft von COs, via externen Interaktionspunkten auf deren angebotene Interfaces zugreifen zu können bzw. Interfacereferenzen für deren genutzte Interfaces zu hinterlegen. Entsprechend den Diskussionen in Abschnitt 2.2 ist für den Zugriff auf angebotene Interfaces zwischen der Realisierung für den operationalen und nicht-operationalen Anteil des Interfacetyps im Entwurfsmodell zu unterscheiden (Produktion von zwei unterschiedlichen **provide**-Operationen), während die Hinterlegung im Kontext eines Operationsrufes (**connect**-Operation) erfolgt. Darüber hinaus ist sowohl die Nutzung der generischen, d.h. bereits von der Basisinterfacedefinition **::ComponentModel::ComponentBase** angebotenen Navigationsoperationen (**provide**, **provide_supply** und **connect**) als auch der CO-Typ-spezifischen Operationen möglich. Die mit der Interfacenavigation in Zusammenhang stehenden Interaktionsabläufe werden im folgenden vorgestellt.

Die Schritte während der Interfacenavigation von einem CO zum operationalen Anteil eines an einer *Provided-Port*-Definition angebotenen Interfaces unter Nutzung der CO-Typ-spezifischen Operationen für die in *Beispiel 43* präsentierte Situation sind in Abb. 33 dargestellt. Die Umgebung nutzt dazu die *Provided-Port*-spezifische Operation **provide_<Port-Name>** (vgl. *Regel 26*). Diese CORBA-IDL-Operation wird durch die

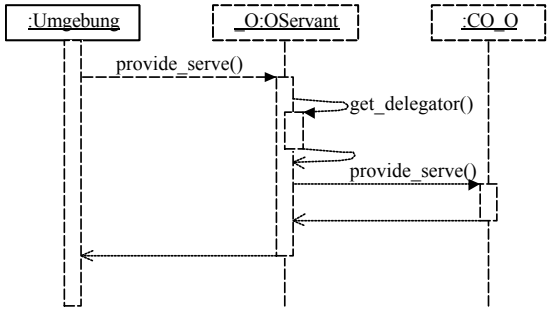


Abb. 33 Interfacenavigation unter Nutzung der CO-Typ-spezifischen Operationen

Klasse **OServant** durch Delegierung an die Klasse **OComposition** realisiert (vgl. *Regel 48*), die wiederum Basisklasse für **CO_O** ist (vgl. *Regel 44*). Die Referenz auf die bei der Konstruktion der **OServant**-Instanz übergebene **OComposition**-Instanz wird durch die Methode **get_delegator** zurückgegeben (vgl. *Regel 41*). Die Implementierung von **provide_serve** der Klasse **OComposition** gibt die der *Port*-Definition ***serve*** entsprechende *Servant*-Instanz der Klasse **iServant** zurück, die gerade durch sie selbst erzeugt wurde (vgl. *Regel 48*). Der Rufer von **provide_serve** in der Umgebung ist nun in der Lage, die an der *Provided-Port*-Definition **serve** bereitgestellte Interfacereferenz vom Typ ***i*** zu nutzen.

Falls der Nutzer der Interfacenavigation in der Umgebung des COs vom Typ ***O*** die durch **::ComponentModel::ComponentBase** definierte generische Operation **provide** unter Angabe des *Provided-Port*-Namens **"serve"** als Parameter benutzt, ergibt sich der in Abb. 34 dargestellte Ablauf. Dabei werden durch die Implementierung der Klassenmethoden von **OServant** die generischen **provide**-Operationen durch die CO-Typ-spezifischen Methodenimplementierungen ausgeführt.

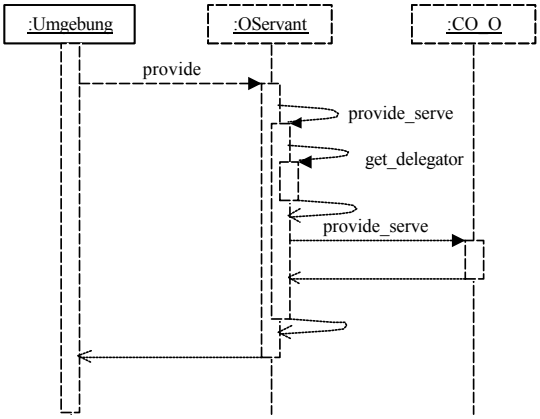


Abb. 34 Interfacenavigation unter Nutzung der generischen Operationen von ::ComponentModel::ComponentBase

Die Realisierung der Interfacenavigation für den nicht-operationalen Anteil von Interfacetypen im Entwurfsmodell hat neben der eigentlichen Rückgabe einer Interfacereferenz für das diesem Anteil zugeordnete CORBA-Interface die Konfiguration der Kommunikationskanäle für Signal- und *Continuous-Media*-Interaktionselemente zur Aufgabe. Während die Konfiguration für *Continuous-Media*-Interaktionen einzig durch Ruf von Managementoperationen an den im Rückgabewert referenzierten **MediaManager**- bzw. **MediaDevide**-Objekten durch $CORE_{WARE}$ erfolgt, wird zur Konfiguration der Signalkommunikationskanäle das durch den *Container* bereitgestellte Management von CORBA-*Event*- bzw. CORBA-*Notification*-Kanälen genutzt. Der Darstellung in Abb. 35 liegt der Fall zugrunde, daß die Interfacenavigationsoperation **provide_<Port-Name>__Supply** ohne Instanzen von **ChannelIdentification** für die im Interfacetyp im Entwurfsmodell definierten *Produce*- bzw. *Consume*-Interaktionselemente gerufen wird. In diesem Fall werden entsprechend *Regel 67* CORBA-*Event*-Kanäle instanziiert und deren Identifikation in die **inout**-Liste der Parameter der **provide_<Port-Name>__Supply**-Methode aufgenommen. Für die in *Beispiel 43* dargestellte Situation ergibt sich unter diesen Voraussetzungen gerade der in Abb. 35 illustrierte Ablauf. Dabei ruft die Umgebung eines COs (i.allg. $CORE_{WARE}$ selbst bzw. ein angeschlossenes Konfigurationswerkzeug) die Operation **provide_serve__Supply** an einer CORBA-Objektreferenz vom Typ der CORBA-Interfacedefinition **O**, das den CO-Typ nach außen repräsentiert. Durch die **OServant**-Instanz wird dieser Ruf analog zur Navigation für operationale Interfacetypen an die Klasseninstanz vom Typ **CO_O** delegiert. Die Implementierung der Methode **provide_serve__Supply** ruft eine Instanz der Klasse **i__Supplier_Impl**, deren Konstruktor die Konfiguration für *Produce*-Interaktionselemente übernimmt. Die entsprechend *Regel 66* produzierte Klassenmethode **notify_connect_a2_sig_out** der Artefaktrepräsentation für ***A2*** wird zur Benachrichtigung über den Anschluß der instanziierten Signal-*Producer*-Klasse an den CORBA-*Event*-Kanal gerufen. Nachfolgend wird die Konfiguration für *Consume*-Interaktionselemente ausgeführt und die CORBA-Objektreferenz vom Typ **i__Supply__::I__Supply** zurückgegeben.

Die Implementierung der **connect**-Operation zur Hinterlegung von CORBA-Objektreferenzen für genutzte Interfacetypen realisiert sowohl die Konfiguration des operationalen Teils des Interfacetyps als auch die seines nicht-operationalen Teils. Dabei wird die CORBA-Objektreferenz vom Typ des CORBA-IDL-Interfaces, das den operationalen Teil repräsentiert, mittels der Klassenmethode **set_<Port-Name>** hinterlegt. Anschließend wird für alle durch ein CO potentiell produzierten Signale (d.h. für *Consume*-Definitionen an dem Interfacetyp im Entwurfsmodell) die Klasse **<Interfacename>__Use__::<Interfacename>_Supplier_Impl** instanziiert, die die Konfiguration der CORBA-*Event*-Kanäle vornimmt und die durch sie instanziierten Instanzen der Signal-*Producer*-Klassen mit diesen Kanälen verknüpft. Anschließend werden an den die *Consume*-Interaktionselemente implementierenden Artefaktrepräsentationen die entsprechenden Methoden

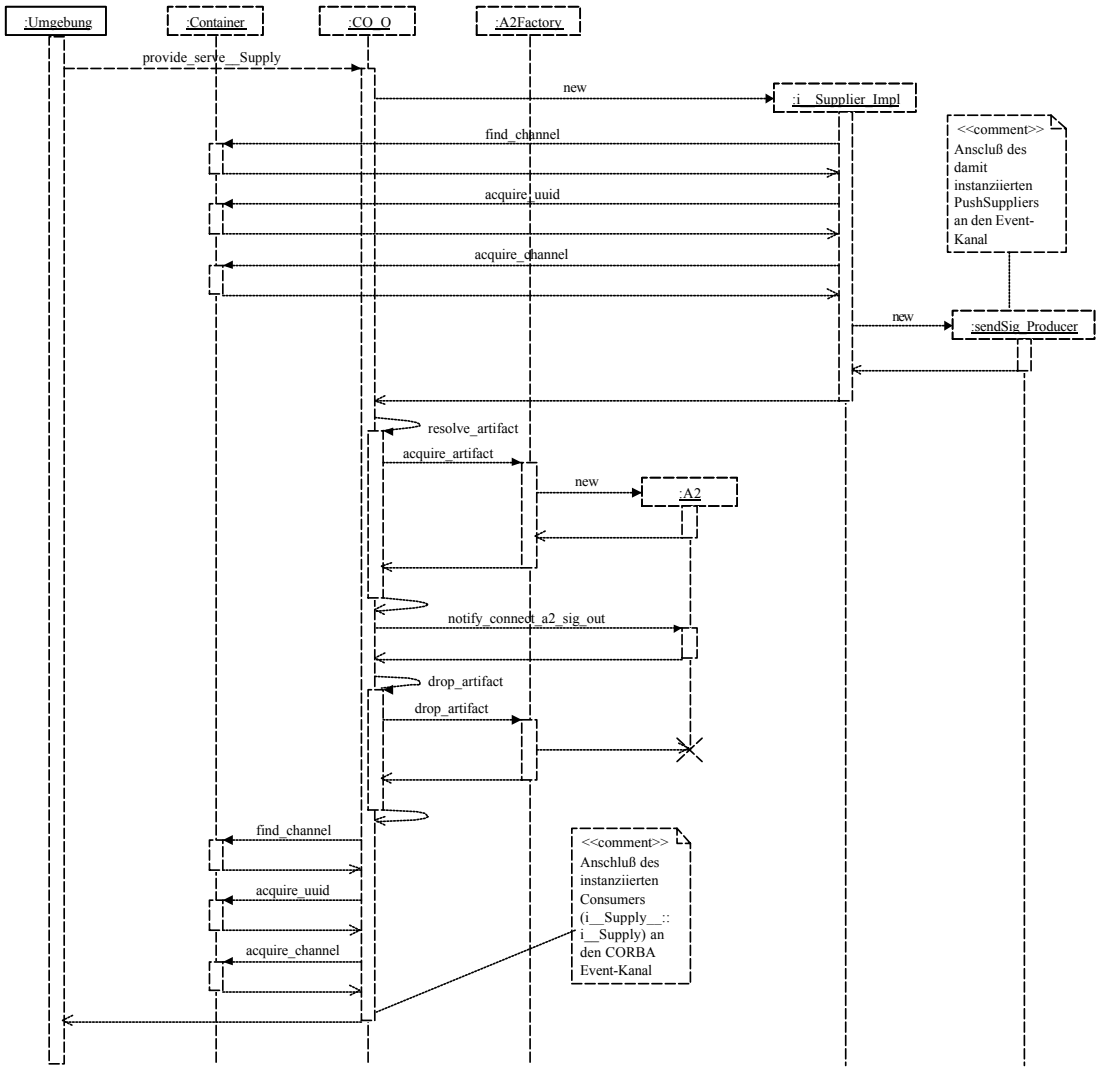


Abb. 35 Realisierung nicht-operationaler provide-Operationen

notify_connect_<Implementierungselement> gerufen und die Konfiguration für durch ein CO konsumierte Signale (*Produce*-Definitionen) vorgenommen. Abschließend wird die CORBA-Objektreferenz des Typs **<Interfacename>__Use__::<Interfacename>__Use** zurückgegeben. Den Aufruf von **connect_use** für die *Port*-Definition mit dem Namen ***use*** aus *Beispiel 43* verdeutlicht Abb. 37.

3.5 Nutzung von angebotenen Interfaces - Aufruf von Operationen

Für die in *Beispiel 43* dargestellte Situation kann die Umgebung eines COs vom Typ ***O*** die an der *Provided-Port*-Definition ***serve*** bereitgestellte Interfacereferenz vom Typ ***i*** nutzen, um beispielsweise die in ***i*** definierte Operation ***op*** zu rufen. Der Ablauf der Behandlung des Operationsrufes ***op*** im Kontext der Repräsentation des COs ist in Abb.38 dargestellt. Der von der Umgebung initiierte Operationsruf wird durch die **iServant**-Instanz entgegengenommen. Mit Hilfe der Operation **get_delegator** erhält diese Instanz eine Referenz auf die entsprechende CO-Typ-*Composition*-Klasseninstanz - und mithin auf deren Basisklasse vom Typ **iComposition**. Diese Referenz wird genutzt, um zunächst die die Operation **op** behandelnde

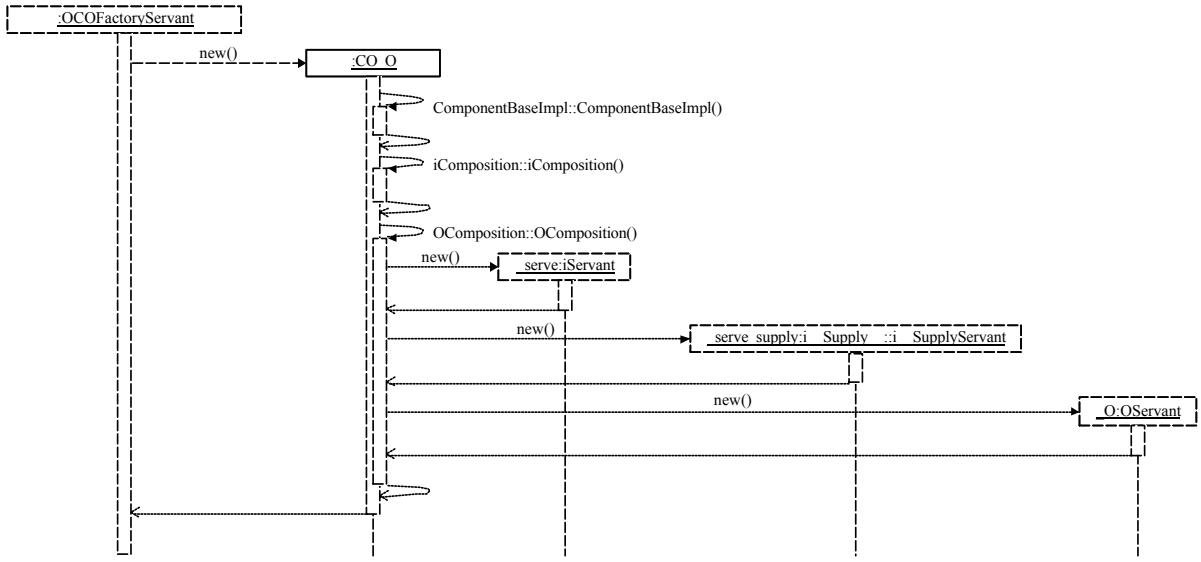


Abb. 36 Erzeugung von Composition- und Servant-Klasseninstanzen

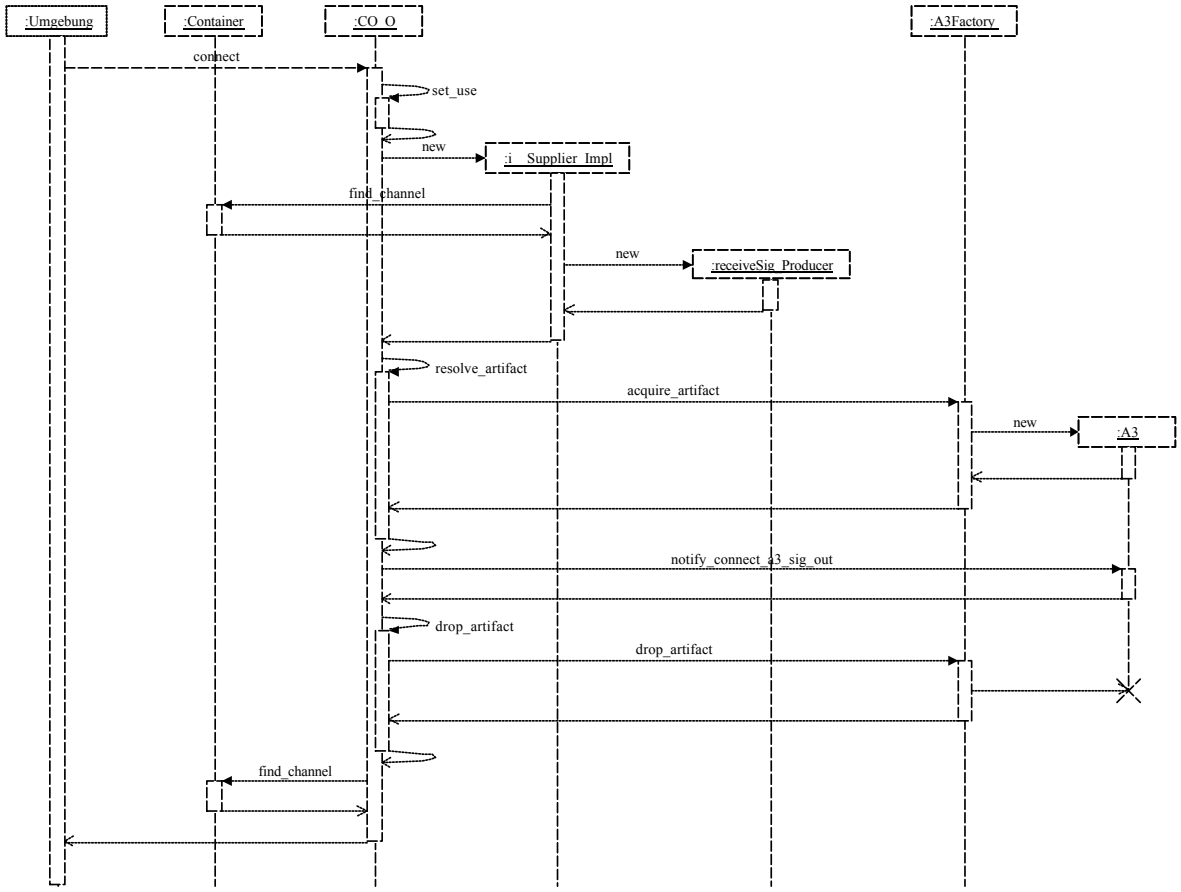


Abb. 37 Realisierung von connect-Operationen

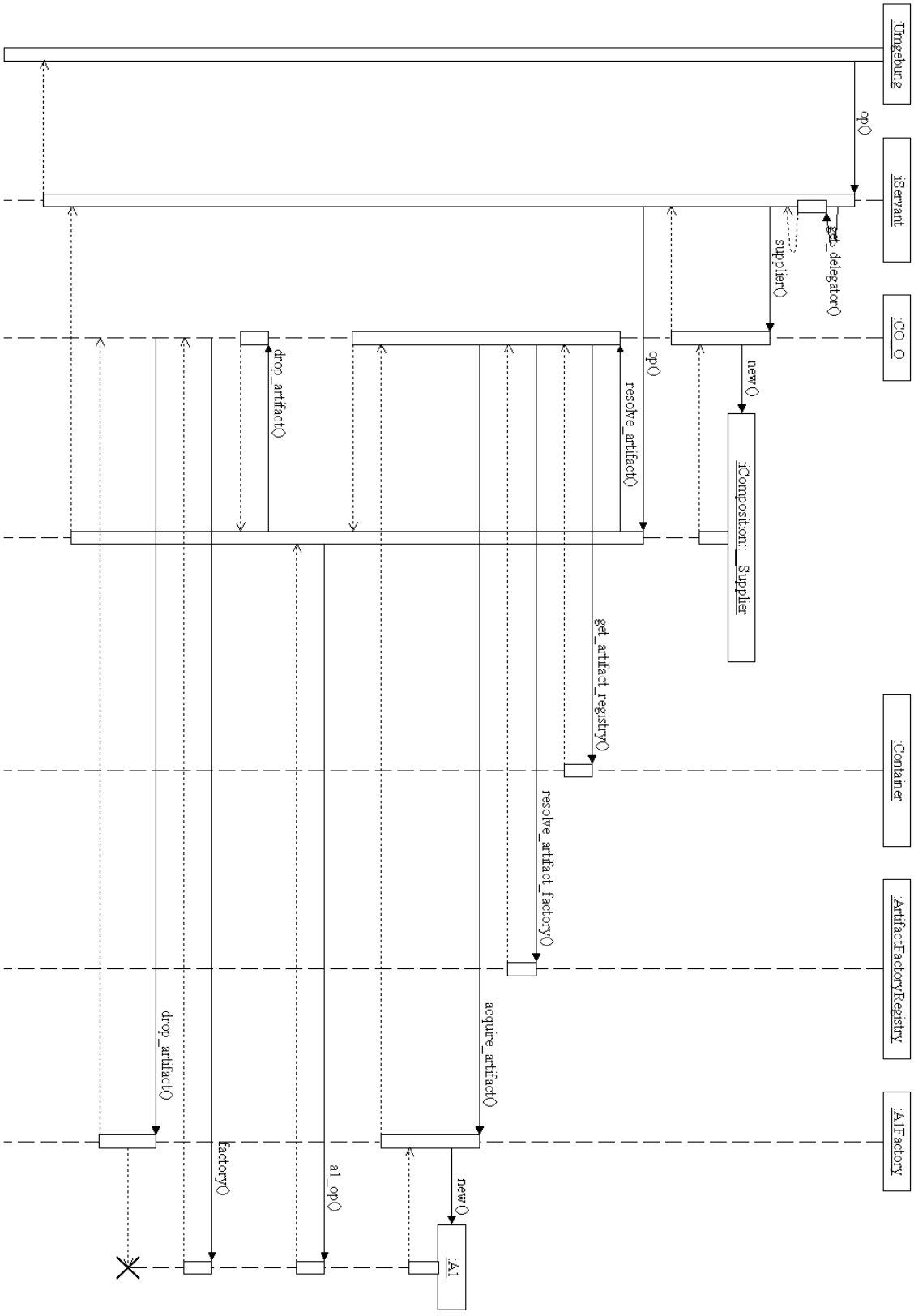


Abb. 38 Ablauf eines Operationsrufes

iComposition::__Supplier-Klasseninstanz via **supplier()** zu erhalten. In diesem Beispiel wird angenommen, daß die an der *Provided-Port*-Definition ***serve*** bereitgestellte Interfacereferenz vom Typ ***i*** erstmalig genutzt wird. Dementsprechend erzeugt die produzierte Implementierung von **supplier()** eine Instanz der Klasse **iComposition::__Supplier** und gibt diese zurück. Anschließend wird der Operationsruf **op** an diese Instanz delegiert. Durch die **iComposition::__Supplier**-Klasseninstanz wird zunächst mittels der Methode **resolve_artifact** - gerufen an der im Konstruktor übergebenen **iComposition**-Klasseninstanz (vgl. *Regel 41*) - die die Operation **op** implementierende Artefaktklasseninstanz ermittelt. Für dieses Beispiel sei angenommen, daß das Instanziierungsmuster ***ARTIFACT_PER_REQUEST*** vorgegeben ist. Dementsprechend erzeugt die Implementierung der Artefakt-*Factory*-Operation **acquire_artifact** eine Instanz der Artefaktklasse **A1** und gibt diese an die **iComposition::__Supplier**-Klasseninstanz zurück. Nunmehr ermittelt die **iComposition::__Supplier**-Klasseninstanz den Typ der erhaltenen Artefaktklasseninstanz und delegiert den Operationsruf **op** an diese. Nach der Ausführung der Delegierung von **op** wird die Artefaktklasseninstanz durch den Ruf von **drop_artifact** wieder freigegeben, im Fall ***ARTIFACT_PER_REQUEST*** implementiert durch Zerstörung der Instanz. Das Ergebnis des Operationsrufes von **op** wird an die **iServant**-Instanz zurückgegeben, und durch diese an die rufende Umgebung weitergeleitet.

3.6 Nutzung von angebotenen Interfaces - Signalinteraktionen

Zur Signalinteraktion werden CORBA-*Event*- bzw. CORBA-*Notification*-Kanäle eingesetzt, die speziellen Konfigurations- und Nutzungsparadigmen unterliegen. Die Konfiguration derartiger Kanäle wird - unterstützt durch den *Container*-Mechanismus von *CORE*$_{WARE}$ - durch die Implementierung der CO-Typrepräsentation vorgenommen.

Während der Konfiguration werden alle instanziierten Signal-*Producer*-Klasseninstanzen an die durch *CORE*$_{WARE}$ erzeugten CORBA-*Event*- bzw. CORBA-*Notification*-Kanäle angebunden. Nach Abschluß der Konfiguration dieser Kanäle wird die Methode **notify_connect_<Implementierungselement>** an der Artefaktrepräsentation gerufen, die das *Produce*-Interaktionselement realisiert. Es liegt nun in der Verantwortung des Entwicklers, Signale zu versenden. In Abb. 39, die das Produzieren von Signalen für die in *Beispiel 43* beschriebene Situation illustriert, wird ein solcher Mechanismus durch die Instanziierung eines nebenläufigen Programmkonstruktes (*Thread*) mit dem Namen **UserDefinedSignalSender** bereitgestellt. Diesem Konstrukt wird der Zugriff auf eine für das *Produce*-Interaktionselement ***sendSig*** entsprechend *Regel 52* erzeugte Signal-*Producer*-Klasseninstanz erlaubt. Diese wird genutzt, um mittels **push_sendSig** Signalrepräsentationen (Instanzen der CORBA-IDL-**valuetype**-Definition **::M1::Sig**) zu versenden. Die Implementierung von **push_sendSig** wiederum überführt diese Signalrepräsentation in eine CORBA-**any**-Darstellung und liefert diese via Basisklassenmethode **push** von **Container::SignalProducer** an den CORBA-*Event*- bzw. CORBA-*Notification*-Kanal aus. Die Nutzung nebenläufiger Programmkonstrukte ist selbstverständlich nicht zwingend erforderlich, das Versenden von Signalen kann z.B. ebenfalls als Reaktion auf den Empfang eines Signals, den Ruf einer Operation etc. erfolgen. Dieses Beispiel impliziert die Notwendigkeit des Einsatzes von Synchronisationsmechanismen, da natürlich mehr als ein nebenläufiges Programmkonstrukt gleichzeitig auf ein und dieselbe Signal-*Producer*-Klasseninstanz zugreifen kann. Solche Synchronisationsmechanismen wurden unter Nutzung der C++-Klassenbibliothek JTC (*Java Threads for C++*, [OOC JTC]) realisiert[1].

Die Konfiguration von Signalkommunikationskanälen für den Empfang von Signalen (Instanzen der entsprechend *Regel 4* produzierten CORBA-IDL-**valuetype**-Definitionen) erfolgt - analog zur Konfiguration für zu sendende Signalrepräsentationen - durch die CO-Repräsentation selbst. Durch die Ableitungsregeln der Strukturansicht ist impliziert, daß die für nicht-operationale Interaktionselemente produzierten CORBA-IDL-

1. Synchronisationsmechanismen wurden an vielen anderen Stellen eingesetzt, um Konflikte beim parallelen Zugriff auf Ressourcen zu vermeiden, so z.B. beim Zugriff auf Artefaktrepräsentationen, während der Initialisierung des Containers etc.

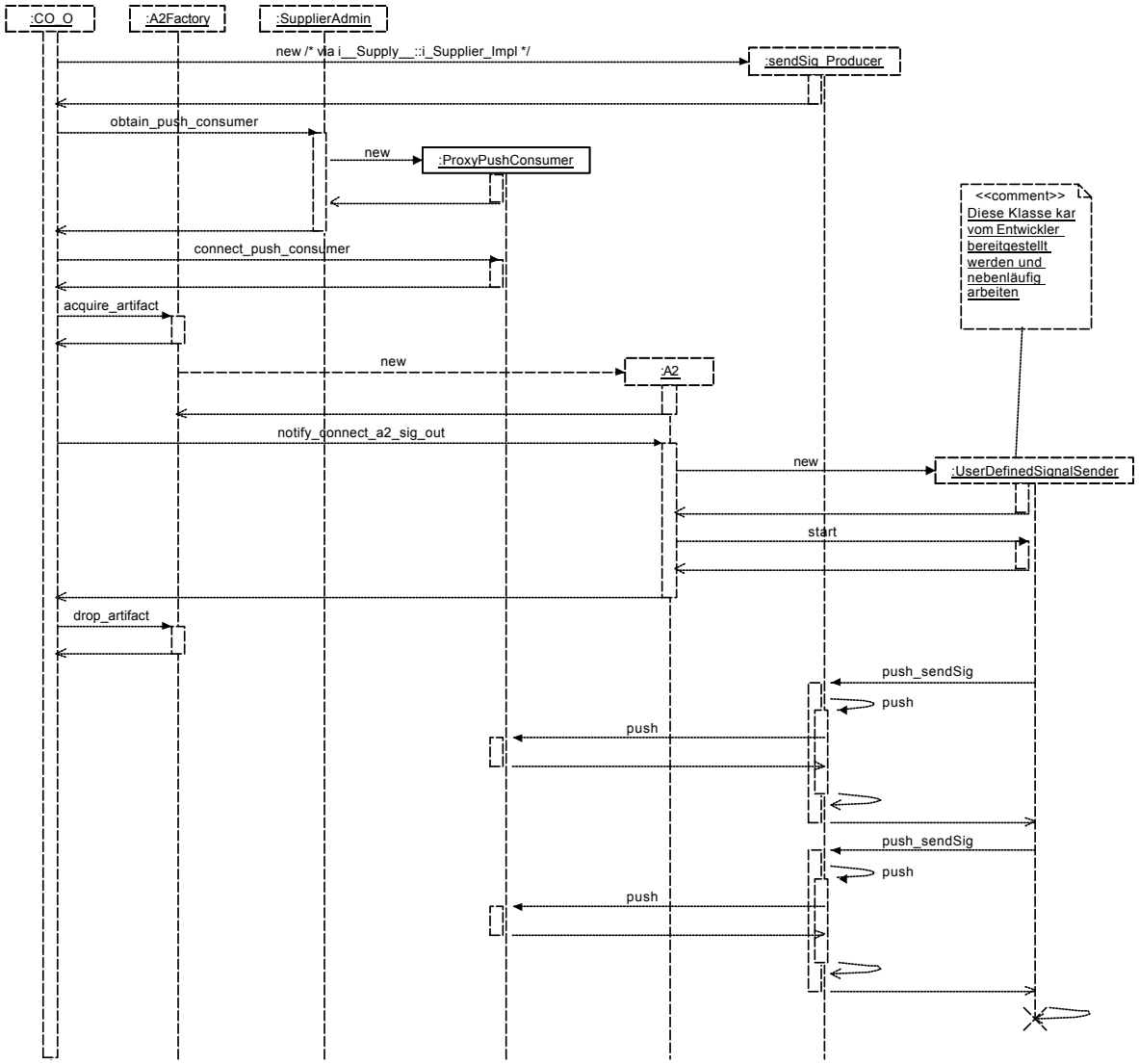


Abb. 39 Versenden von Signalen durch vom Entwickler bereitgestellte Implementierungselemente

Interfacedefinitionen **<Interfacename>__Supply__::<Interfacename>__Supply** bzw. **<Interfacename>__Use__::<Interfacename>__Use** Spezialisierungen der CORBA-IDL-Interfacedefinitionen sind, die zum Empfang von Signalen (für jedes *Consume*-Interaktionselement im *Supports*-Fall bzw. jedes *Produce*-Interaktionselement im *Requires*-Fall) erzeugt wurden. Die Konfiguration dieser Signalempfänger beschränkt sich demzufolge auf die Kopplung der *Servant*-Klasseninstanz (**<Interfacename>__Supply__::<Interfacename>__SupplyServant** bzw. **<Interfacename>__Use__::<Interfacename>__UseServant**) mit einem CORBA-*Event*- bzw. *Notification*-Kanal. Der Empfang von Signalen bedeutet dann die Auslieferung eines Signals (d.h. Instanz der korrespondierenden CORBA-IDL-**valuetype**-Repräsentation) durch den Signalkommunikationskanal an die angeschlossene *Servant*-Klasseninstanz, Ableitung der getypten Repräsentation aus der CORBA-**any**-Repräsentation des Signals und Auslieferung dieser Repräsentation an die zugehörige Artefaktklasseninstanz. Die Konfiguration dieser Kanäle ist für die in *Beispiel 43* dargestellte Situation in Abb. 40 illustriert. Das Szenario setzt voraus, daß die korrespondierenden CORBA-*Event*- bzw. CORBA-*Notification*-Kanäle bereits durch *CORE*$_{WARE}$ instanziiert wurden, so daß die Liste der Konfigurationsparameter für den Ruf der Operation

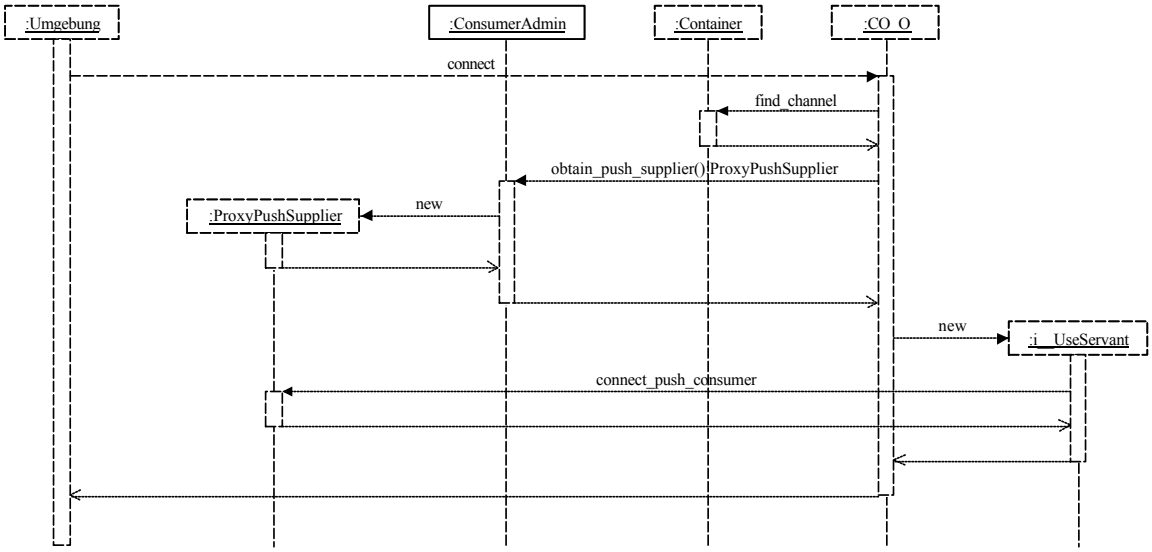


Abb. 40 Konfiguration von Signalkommunikationskanälen auf Empfängerseite

connect_<Port-Name> bereits Instanzen von **ChannelIdentification** für alle entsprechenden Kanäle enthält. Dementsprechend wird während der Hinterlegung einer genutzten Interfacereferenz des Typs ***i*** an einer CO-Repräsentation vom Typ ***O*** für *Beispiel 43* eine Instanz der Klasse **i__Use__::i__UseServant** erzeugt und mit einem CORBA-*Event*- bzw. *Notification*-Kanal durch Ruf der Operation **connect_push_consumer** an einem CORBA-Objekt vom Typ **CosEventChannelAdmin::ProxyPushSupplier** verbunden. Nachfolgend werden durch diesen Kanal - repräsentiert durch das CORBA-Objekt vom Typ **CosEventChannelAdmin::ProxyPushSupplier** - Signalrepräsentationen an die *Servant*-Klasseninstanz ausgeliefert. Dieser Interaktionsablauf ist in Abb. 41 für das *Produce*-Interaktionselement ***sendSig*** im Falle der Nutzung von ***i*** durch ein CO vom Typ ***O*** dargestellt. Zunächst wird die CORBA-**any**-Repräsentation der CORBA-IDL-**valuetype**-Instanz vom Typ **::M1::sendSig** (vgl. *Regel 5*) durch das **ProxyPushSupplier**-Objekt an das durch **i__Use__::i__UseServant** repräsentierte **PushConsumer**-Objekt mittels der Operation **push** ausgeliefert. Die Implementierung von **push** im Kontext dieser *Servant*-Klasse überführt die CORBA-**any**- in die getypte CORBA-IDL-Repräsentation vom Typ **::M1::sendSig** (vgl. *Regel 51*). Anschließend ruft sie die für dieses Interaktionselement erzeugte *Servant*-Klassenmethode **push_sendSig** (vgl. *Regel 51*). Die *Servant*-Klassenmethode **push_sendSig** ermittelt zunächst die Instanz der in der Interface-*Composition*-Klasse **iComposition** enthaltenen lokalen Klasse **__User** mittels **user()**, die beim ersten Aufruf von **user()** erzeugt wird (*Regel 42*). Anschließend nutzt sie das Delegierungsmuster (vgl. [GHJ+ 99]) an diese Instanz zur Realisierung von **push_sendSig** durch Aufruf der gleichnamigen Klassenmethode der Klasse **iComposition::__User**. Die Realisierung dieser Methode löst die dem Interaktionselement ***sendSig*** im *Use*-Fall zugeordnete Artefaktklasse ***A3*** auf und instanziiert diese mittels der korrespondierenden Artefakt-*Factory*-Klasseninstanz (vgl. *Regel 62*). Die das Implementierungselement ***a3_sig_in*** repräsentierende Klassenmethode der resultierenden Artefaktklasseninstanz wird zur Bearbeitung des Signals aufgerufen. Anschließend wird die Nutzung der Artefaktklasseninstanz mittels **drop_artifact** beendet.

In der Kombination der Interaktionsabläufe für das Produzieren und Konsumieren von Signalen ergibt sich der in Abb. 42 dargestellte Gesamtablauf. Die illustrierte Situation ergibt sich, falls ein CO vom Typ ***O*** ein Interface ***i*** an einer *Provided-Port*-Definition ***serve*** anbietet, das von einem weiteren CO desselben Typs via *Used-Port*-Definition ***use*** genutzt wird (vgl. *Beispiel 43*). Der Anbieter von ***i*** produziert entsprechend der Definition des *Produce*-Interaktionselements ***sendSig*** Signale des Typs ***Sig***, die vom Nutzer desselben Interfacetyps empfangen werden. Dabei wird die Produktion eines derartigen Signals auf die Auslieferung der CORBA-**any**-Repräsentation desselben an einen CORBA-*Event*- bzw. *Notification*-Kanal abgebildet. Der Transport die-

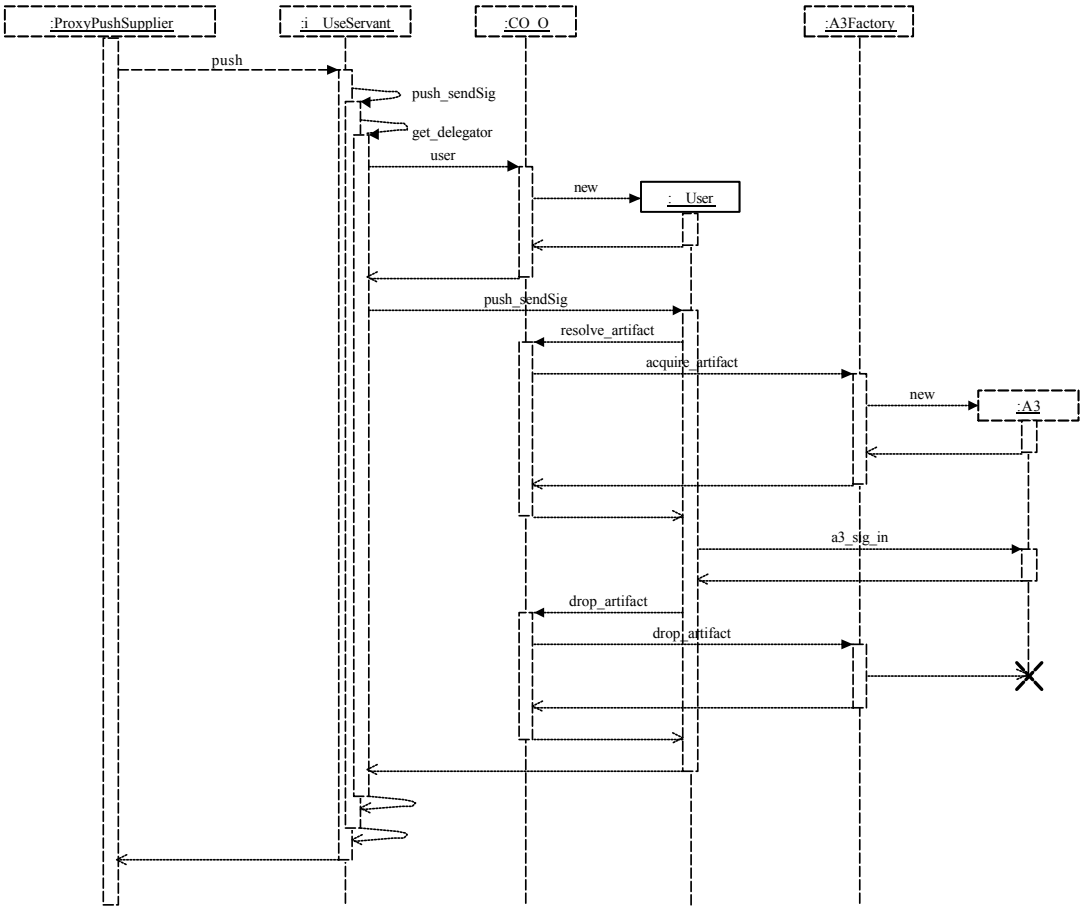


Abb. 41 Signalauslieferung

ser Repräsentation erfolgt innerhalb des Kanals i.allg. asynchron, d.h. der Auslieferer des Signals erhält die Programmablaufkontrolle sofort zurück. Der Kanal wiederum liefert die CORBA-**any**-Repräsentation an das durch **`i__UseServant`** repräsentierte CORBA-Objekt vom Typ **`CosEventComm::PushConsumer`** aus. Die *Servant*-Klassenimplementierung delegiert die Bearbeitung des Interaktionselements an die Realisierung der Interface-*Composition*-Klasse, die letztlich für den Aufruf des korrespondierenden Implementierungselements in der Artefaktrepräsentation verantwortlich ist.

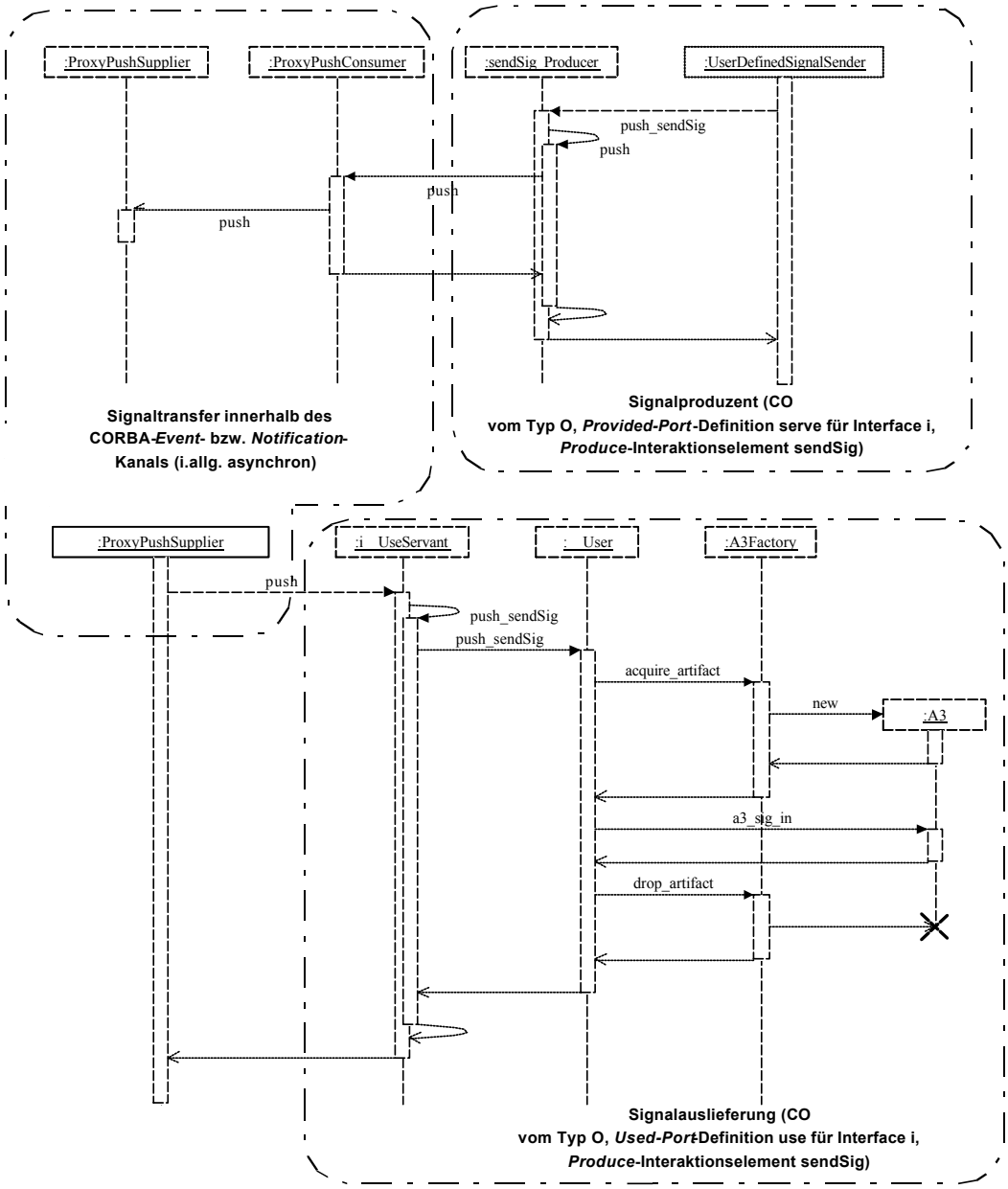


Abb. 42 Interaktionsablauf für Signale

KAPITEL 4

Resümee und Ausblick

In diesem Band wurden die Entwicklungstechniken Einsatz von Komponentenarchitekturen sowie die automatische Ableitung von Softwarekomponenten aus Entwurfsmodellen im Kontext von *CoRE* diskutiert.

Zusammenfassend kann festgestellt werden, daß für *alle* Konzepte des Konzeptraumes von *CoRE* Ableitungsregeln für CORBA-IDL-Interfacedefinitionen und programmiersprachliche Konstrukte formuliert wurden. Die Basiskonzepte des Konzeptraumes, namentlich das Konzept Interfacetyp als Interaktionskontext für unterschiedliche Interaktionsarten und das Konzept CO-Typ als Kontext des Anbietens bzw. Benutzens von Interfaces, wurden durch Ableitungsregeln zur Erzeugung von CORBA-IDL bzw. C++ umgesetzt. Insbesondere die Definition von Ableitungsregeln zur Erzeugung von Interface- bzw. CO-Typ-*Composition*-Klassen stellt diese Kontexte innerhalb der internen Sicht auf einen CO-Typ her. Damit erfüllen die Ableitungsregeln die als grundsätzlich dargestellte *Anforderung (17) in Abschn. 1.1.9* (Vollständigkeit der Ableitungsregeln). Die produzierten CORBA-IDL- und C++-Definitionen reflektieren - für einen Entwickler nachvollziehbar - die Struktur der Definitionen für einen CO-Typ und einen Interfacetyp in einem Entwurfsmodell.

Die Ableitungsregeln basieren auf $CoRE_{WARE}$ die - aufbauend auf existierenden Technologien wie CORBA 2.4 und Aspekten von CORBA *Components* - eine Infrastruktur zur *Deployment*- und Ausführungsunterstützung von Softwarekomponenten realisiert. $CoRE_{WARE}$ wurde durch den Autor prototypisch implementiert. Als Basis der Implementierung dieser Plattform dienten die CORBA-Produkte *ORBacus* [OOC] und *MICO* [MICO]. $CoRE_{WARE}$ realisiert die Kommunikationsinfrastruktur für Repräsentationen von CO-Typen:

- Signalinteraktionen werden durch von $CoRE_{WARE}$ bereitgestellte und verwaltete CORBA-*Event*- bzw. *Notification*-Servicekanäle realisiert,
- *Continuous-Media*-Interaktionen durch die Integration der an der Humboldt-Universität zu Berlin unter Beteiligung des Autors entstandenen *Continuous-Media-Delivery*-Plattform in $CoRE_{WARE}$ und
- operationale Interaktion durch CORBA-Mechanismen selbst.

Die Plattform implementiert das *Container*-Konzept als Ausführungsumgebung für CO-Typ-Repräsentationen. $CoRE_{WARE}$ definiert eine einheitliche Sicht auf CO-Typen und deren Instanzen durch Spezifikation von

CORBA-IDL-Interfacedefinitionen, die durch die Repräsentationen von CO-Typen spezialisiert werden - eine wesentliche Grundlage für die Erfüllung der Offenheitsanforderung *in Abschn. 1.1.1*. Mit der Realisierung von $CoRE_{WARE}$ wurde die prinzipielle Machbarkeit der Abbildung der Elemente von $CoRE_{CEPT}$ auf Mechanismen einer *Component-Support*-Plattform nachgewiesen und eine einheitlichen Sicht auf die atomaren Bestandteile eines verteilten Softwaresystems verwirklicht.

Durch die Definition der externen Sicht auf einen CO-Typ und dessen Instanzen sowie die Realisierung des *Port*-Managements im Kontext der internen Sicht auf einen CO-Typ ist es gelungen, das Prinzip des generischen Konfigurationsmanagements zu unterstützen. Es konnte sowohl das Konfigurationsmanagement für *Port*-Definitionen für rein operationale Interfacetypen realisiert werden als auch für Interfacetypen, die einen Kontext für Interaktionselemente unterschiedlicher Interaktionsarten bilden. Dazu wurden in $CoRE_{WARE}$ Mechanismen zur Bereitstellung und Steuerung von Kommunikationskanälen für Signal- und *Continuous-Media*-Interaktionen realisiert. Das Prinzip des generischen Konfigurationsmanagements wurde durch die Erstellung eines Konfigurationswerkzeuges demonstriert (vgl. Abschnitt 2.4). Im Rahmen des EURESCOM Projektes P924 wurde auf der Basis von $CoRE_{WARE}$ und $CoRE_{MAP}$ ein Beispieldienst realisiert, der mittels desselben Prinzips durch in diesem Projekt entwickelte *Deployment*- und Konfigurationswerkzeuge automatisch konfiguriert werden kann [BSH 01]. Der Beispieldienst selbst wurde mit den in $CoRE_{WARE}$ integrierten, auf der Grundlage der Ableitungsregeln für Softwarekomponenten realisierten Werkzeuge implementiert [BK 01][BK 01a]. Durch die Projektergebnisse des EURESCOM Projektes P924, die auf der hier vorgestellten *Component-Support*-Plattform und der die Ableitungsregeln von $CoRE_{MAP}$ realisierenden Werkzeugkette basieren, wurde nachgewiesen, daß die *Anforderung (1) in Abschn. 1.1.1* (Konfigurationsmanagement und Navigation) erfüllt werden konnte.

Die Verfügbarkeit von Entwurfsinformationen zur Ausführungszeit ist durch die Verfügbarkeit von Entwurfsmodellen gegeben, die mittels *CoRE* erstellt wurden. Diese Entwurfsmodelle können mittels XMI [OMG XMI 1.1] zwischen verschiedenen Werkzeugen ausgetauscht werden. Eine weitere, bisher nicht diskutierte bzw. realisierte Möglichkeit der Bereitstellung von Entwurfsinformationen zur Ausführungszeit besteht in der Anwendung der durch *Meta Object Facility* (MOF, [OMG MOF1.3]) definierten Ableitungsregeln für die Erzeugung von CORBA-IDL-Definitionen aus Metamodellen. Diese Technologie ermöglicht die automatische Generierung von CORBA-IDL-Interfacedefinitionen und deren Implementierungen, die gerade Informationen über konkrete Entwurfsmodellelemente verwalten, die auf der Basis von $CoRE_{CEPT}$ definiert wurden. Mit dieser Technologie kann ein Analogon des CORBA-Standards *Interface Repository* für CO-Typen, deren *Port*-Definitionen, interne Struktur etc. geschaffen werden. Beide Lösungsansätze - Entwurfsmodellaustausch selbst bzw. Nutzung der MOF-Generierungsregeln für CORBA-IDL - erfüllen die *Anforderung (2) in Abschn. 1.1.1* bezüglich der Reflexion von Entwurfsinformationen zur Ausführungszeit.

Artefaktrepräsentationen sind wichtige Elemente der Wiederverwendung von Softwarekomponenten. Sie beinhalten diejenigen durch den Entwickler hinzugefügten programmiersprachlichen Konstrukte, die das spezifische Verhalten von CO-Typen erbringen. Die Ableitungsregeln wurden so definiert, daß Artefaktrepräsentationen *de facto* keine Abhängigkeiten von den Mechanismen in $CoRE_{WARE}$ enthalten. Die Herstellung des Kontextes eines CO-Typs und dessen Relationen zu Artefaktrepräsentationen wird durch das in der CO-Typ-*Composition*-Klasse realisierte Artefaktmanagement erreicht. So können durch einen Entwickler vollständig implementierte Artefaktklassen im Kontext verschiedener CO-Typen wiederverwendet werden. Artefaktklassen spezialisieren die C++-Klasse **`Container::Artifact`**, die die Sicht von $CoRE_{WARE}$ auf Artefaktklassen und deren Instanzen reflektiert und im *Container*-Kontext implementiert ist. Die Notwendigkeit der Spezialisierung dieser Klasse ist die einzige konkrete Vorgabe bezüglich der Gestaltung von Artefaktklassen. Die Anbindung der Instanzen der Artefaktklassen an die Mechanismen von $CoRE_{WARE}$ wird durch das Artefakt- und Interaktionsmanagement der CO-Typ- bzw. Interface-*Composition*-Klassen erreicht. Somit erfüllen die Ableitungsregeln die *Anforderung (3) in Abschn. 1.1.1* (Unabhängigkeit der Artefaktklassen).

Alle definierten Ableitungsregeln basieren auf den Definitionen des Metamodells von $CoRE_{CEPT}$, die durch [CoRE II], Kapitel3 gegeben sind. Dementsprechend impliziert eine Erweiterung bzw. Anpassung dieses Metamodells ebenfalls die Erweiterung bzw. Anpassung der Ableitungsregeln für Softwarekomponenten.

Diese Erweiterungen bzw. Anpassungen erfolgen durch Definition weiterer Ableitungsregeln auf der Basis der in $CoRE_{MAP}$ bereits definierten Regeln. Aufgrund der nicht-rekursiven Abhängigkeiten zwischen den Regeln ist diese Form der Erweiterung bzw. Anpassung einfach möglich. Natürlich müssen Werkzeuge, die die Ableitungsregeln konkret implementieren, ebenso angepaßt werden, jedoch ist der dazu nötige Aufwand abhängig von der Struktur der Implementierung dieser Werkzeuge. Die entwickelte Werkzeugumgebung, die $CoRE_{MAP}$ realisiert, basiert auf einer Menge von C++-Klassen, die den Modellelementen des Metamodells entsprechen und über diesen operierenden Generatorklassen, die die Ableitungsregeln selbst implementieren. Diese Strukturierung erwies sich als einfach erweiterbar und gleichzeitig wartbar. Insofern ist die Erfüllung der *Anforderung (4) in Abschn. 1.1.2* (Erweiterbarkeit von $CoRE_{MAP}$) konzeptionell durch die Definitionen des Metamodells und darauf aufbauende, nicht-rekursive Definitionen von Ableitungsregeln erfüllt. Die prototypische Implementierung ist so gestaltet, daß Erweiterungen bzw. Anpassungen der Ableitungsregeln auf einfache Art und Weise in diese eingebracht werden können.

Die in der Anwendungsdomäne relevanten Interaktionsarten operationale Interaktion, Signalinteraktion und *Continuous-Media*-Interaktion können in Entwurfsmodellen genutzt und durch Ableitung von Softwarekomponenten realisiert werden. Die notwendigen Kommunikationskanäle der Infrastruktur werden durch $CoRE_{WARE}$ selbst bereitgestellt und verwaltet, sie basieren auf Mechanismen der *Distributed-Processing*-Umgebung innerhalb von $CoRE_{WARE}$, die in die *Component-Support*-Plattform integriert wurden. Während für die Unterstützung der Signalinteraktion auf konkrete Produkte, die standardisierte Lösungen implementieren, zurückgegriffen werden konnte, wurde die Unterstützung der *Continuous-Media*-Interaktion basierend auf einer nicht-standardisierten Lösung realisiert. Zum gegenwärtigen Zeitpunkt ist eine standardisierte Lösung für *Continuous-Media*-Interaktionen nicht verfügbar. Falls eine solche entstehen sollte, ist ihre Integration prinzipiell durchführbar, da alle konkreten Mechanismen durch die *Component-Support*-Plattform $CoRE_{WARE}$ abstrahiert werden. Damit ist die *Anforderung (5) in Abschn. 1.1.3* nach der Unterstützung aller relevanten Interaktionsarten erfüllt.

Eine wesentliche Anforderung an die Ableitungsregeln besteht in der Bereitstellung flexibler Mechanismen für skalierbare Softwarekomponenten (vgl. *Anforderung (6)* und *Anforderung (7) in Abschn. 1.1.4*). Es wurde festgestellt, daß der Kernmechanismus für die Flexibilität von Softwarekomponenten bezüglich Skalierbarkeit die Ableitung von programmiersprachlichen Konstrukten für das Konzept Instanziierungsmuster ist. Konkrete Instanziierungsmuster wurden unter Nutzung des Entwurfsmusters *Generic Factory* [GHJ+99] im Kontext von Artefakt-*Factory*-Klassen und deren produzierten Implementierungen sowie im Artefaktmanagement der CO-Typ-*Composition*-Klassen in konkrete programmiersprachliche Konstrukte umgesetzt. Natürlich ist die richtige Auswahl eines bestimmten Instanziierungsmusters durch einen Entwickler die Basis für letztendlich skalierbare Softwarekomponenten. Die Auswahl von Instanziierungsmustern wird im Kontext eines Entwurfsmodells vorgenommen. Die Formulierung „vernünftiger" Richtlinien für die Verwendung von Instanziierungsmustern ist einem Entwickler einfacher vermittelbar als die konkret anzuwendende Implementierungstechnologie zur Umsetzung derselben. Die Ableitungsregeln für Softwarekomponenten sind auch insofern vollständig, als daß durch sie bereits ohne Bereitstellung der Implementierung der Artefaktklassen durch einen Entwickler *ausführbare* Softwarekomponenten entstehen. Es kann während der Entwicklung einer Softwarekomponente durch Angabe verschiedener Instanziierungsmuster in Entwurfsmodellen so lange experimentiert werden, bis entsprechende *Performance*-Analysen den Skalierbarkeitsanforderungen an die zu entwickelnden Softwarekomponenten gerecht werden, ohne daß *Business Logic* integriert werden muß.

Neben der Umsetzung der Instanziierungsmuster ist die möglichst effiziente Anbindung der Artefaktrepräsentationen an die CO-Typrepräsentationen bzw. an die *Component-Support*-Plattform für die Erfüllung der Forderung nach flexibler Skalierbarkeit von Softwarekomponenten essentiell (vgl. *Anforderung (7) in Abschn. 1.1.4*). In Abschnitt 2.3.6 wurde geschlußfolgert, daß die wesentlichen Grundlagen für eine solche flexible Anbindung in nicht-polymorphen Zielen der Anwendung des Entwurfsmusters *Delegation* sowie in der möglichst geringen Anzahl notwendiger Kopiervorgänge für Parameter bestehen. Die Ableitungsregeln im Kontext des Interaktions- und Artefaktmanagements wurden entsprechend formuliert.

Zusätzlich besteht die Forderung nach Minimalität der Abhängigkeiten zwischen Artefakt- und Interfacetyprepräsentationen (vgl. *Anforderung (8)* und *Anforderung (9) in Abschn. 1.1.5*). Diese wurde erfüllt, indem die CO-Typ-*Composition*-Klassen die Repräsentationen (Interface-*Composition*-Klassen) der durch diese angebotenen bzw. genutzten Interfacetypen spezialisieren (vgl. Abschnitt 2.3.1). Insofern beschränken sich die Abhängigkeiten zwischen Artefaktrepräsentationen und *Composition*-Klassen auf die Existenz der durch das Artefaktmanagement definierten Basisklassen für Artefaktklassen sowie die Notwendigkeit, daß Artefaktrepräsentationen diese Basisklassen spezialisieren.

Durch die Diskussion der Ableitungsregeln für die Interaktionssicht wurde gezeigt, daß das Prinzip der Bestimmung von Bindungsfällen anhand von in Entwurfsmodellen definierten Prädikaten durch Codemodule in Softwarekomponenten repräsentiert werden kann. Die Bestimmung eines Bindungsfalls ist durch generische Mechanismen von $CoRE_{WARE}$ möglich. Es wurde diskutiert, daß - begründet in einer geringen Anzahl dazu notwendiger Operationsrufsequenzen - die Bestimmung eines Bindungsfalles keine wesentlichen Performanzeinschränkungen impliziert. Die Gütebeschreibung von Interaktionen zwischen COs und deren Sicherstellung und Überwachung wurden durch Ableitungsregeln in Codemodulen von Softwarekomponenten sowie durch die Definition eines Rahmenwerk zur Erfassung von Kontrakten und *Plug-In*-Mechanismen zur Sicherung von Güteeigenschaften zur Ausführungszeit repräsentiert. Diese stellen einen Ansatz zur Erfüllung der Anforderungen bezüglich der Verhandlung und Zusicherung von Dienstgüteeigenschaften (*Anforderung (10)* und *Anforderung (11) in Abschn. 1.1.6*) dar. Jedoch wurde bereits in Abschnitt 2.5 diskutiert, daß im Kontext dieser Arbeit nur initiale Lösungen untersucht wurden, deren Weiterentwicklung in internationalen und nationalen Projekten erfolgt (vgl. [BKvH 00]).

Durch die Unabhängigkeit der Repräsentationen von Artefakten von der *Component-Support*-Plattform ist die Integration von existierenden Codemodulen, die im Quelltext vorliegen, als auch von Softwarekomponenten, die im Binärformat verfügbar sind, in zu entwickelnde Softwarekomponenten möglich. Durch den Abbildungsmechanismus der Implementierungselemente auf Klassenmethoden der Artefaktrepräsentation ist es möglich, die in existierenden Codemodulen vorliegenden Klassen und deren Methoden als Implementierungselemente heranzuziehen. Durch die Flexibilität, die die Entkopplung zwischen Kontext eines Interaktionselements (Interfacetyp) und Kontext eines Implementierungselements (Artefakt) bietet und die dieses Prinzip umsetzenden Ableitungsregeln des Artefakt- und Interaktionsmanagement können Klassenstrukturen bestehender Codemodule nachgebildet werden. Diese Mechanismen sind die Basis zur Erfüllung der *Anforderung (12) in Abschn. 1.1.7* nach Unabhängigkeit der Artefaktrepräsentation von $CoRE_{WARE}$ sowie der *Anforderung (14)* nach Integrierbarkeit vorhandener Softwarekomponenten bei.

Im Kontext der Diskussion der Ableitungsregeln für Zustandsattribute eines CO-Typs wurde bereits auf die Möglichkeit der Integration des durch *Object Management Group* normierten *Persistent-State*-Service [OMG PSS 2.0] eingegangen. Die in dieser Spezifikation enthaltenen CORBA-IDL-Interfacedefinitionen stellen einen Ansatz zur Integration unterschiedlicher Technologien persistenter Datenspeicher dar, so daß durch die Definition der Ableitungsregeln sowie der durch $CoRE_{WARE}$ bereitgestellten Basisklassen für die Anbindung persistenter Speicher die Integration mit dem *Persistent-State*-Service naheliegt. Eine konkrete Realisierung einer solchen Integration mit der prototypischen Implementierung von $CoRE_{WARE}$ wurde bisher nicht vorgenommen, eine Untersuchung der Integrationsmöglichkeiten zeigt jedoch deren Machbarkeit und damit die Erfüllbarkeit von *Anforderung (13) in Abschn. 1.1.7* nach der Integrierbarkeit von persistenten Speichertechnologien.

Durch die *Anforderung (15) in Abschn. 1.1.8* wird gefordert, daß Werkzeuge realisiert werden, die die Ableitung von Softwarekomponenten entsprechend den definierten Ableitungsregeln automatisieren. Eine derartige Werkzeugkette wurde durch den Autor realisiert und zusammen mit der Realisierung von $CoRE_{CEPT}$ und $CoRE_{TATIONS}$ an das Produkt Rational Rose [RationalRose] angebunden. Diese Werkzeugumgebung implementiert *alle* Ableitungsregeln, die für $CoRE_{MAP}$ definiert wurden, so daß die Werkzeugumgebung selbst die Realisierbarkeit von Werkzeugen auf der Basis der definierten Ableitungsregeln nachweist. Diese Werkzeugkette wird in einem aktuellen Projekt ([MDTS 01]) so erweitert, daß auch die *Anforderung (16) in Abschn. 1.1.8* nach Modellaktualisierung und *Round-Trip* gewährleistet wird.

Im Rahmen internationaler Konferenzen zur Softwareentwicklung für verteilte Systeme (vgl. z.B. [BHK 01], [BK 00a] und [BKvH 00]) sowie in Standardisierungsgremien (vgl. z.B. [BK 00b], [BK 01b]) wurden die Ableitungsregeln von $CoRE_{MAP}$ sowie die erweiterte Komponentenarchitektur $CoRE_{WARE}$ begleitend zu deren Entwicklung vorgestellt. Die daraus entstandenen Diskussionen mit zahlreichen Experten haben die prinzipielle Übertragbarkeit auf andere Bereiche der Softwareentwicklung gezeigt, woraus sich konkrete Untersuchungsgegenstände für weitere Arbeiten ergeben.

Der mögliche Einfluß von $CoRE_{MAP}$ und $CoRE_{WARE}$ auf aktuelle oder in Vorbereitung befindliche Standardisierungsprozesse sowie der Einsatz und die Weiterentwicklung von *CoRE* in Industrieprojekten wurden in [CoRE I], Kapitel 4 umfassend dargestellt.

REFERENZEN

[ANSIC++] ISO/IEC 14882:1998: "*Programming languages - C++*", ISO '98

[BF 01] Born, Fischer: "*Object Definition Language*" Telektronikk 04/2001, Kjeller, Norway '01

[BFL+ 99] Born, Fischer, Löwis, Krüger, Ulbricht: "*Service Composition in a TINA Enviroment*", Proceedings of the TINA 1999 Conference, Oahu, USA '99

[BH 98] Born, Hoffmann: "*An Object-Oriented Design Methodology for Distributed Systems*", Proceedings of the Technology of Object-Oriented Languages and Systems 1998 (TOOLS Pacific 1998) Conference, Melbourne, Australia '98

[BH 98a] Born, Hoffmann: "*Advanced Distribution and Configuration Support for Distributed Applications*" Proceedings of the Workshop on Distributed Object Technologies for Telecoms Networks (DOT'98), Heidelberg, Germany '98

[BHK 01] Born, Holz, Kath: "*Manufacturing Software Components from Object-Oriented Design Models*", Proceedings of the 5th International Enterprise Distributed Object Computing Conference (EDOC 2001), Seattle, USA '01

[BHL+ 98] Born, Hoffmann, Li, Schieferdecker: "*Applying a Framework Approach with Validation to the Design of Telecommunication Services*": Proceedings of the International Conference on Communication Technologies (ICCT'98), Beijing, China '98

[BHL+ 99] Born, Hoffmann, Li, Schieferdecker: "*Using Formal Methods for the Design of Telecommunication Services*" Proceedings of the Conference on Formal Methods for Object Oriented Distributed Systems (FMOODS) '99, Florence, Italy '99

[BHS+99] Born, Hoffmann, Schieferdecker, Vassiliou-Gioles, Winkler: "*Performance Testing of a TINA Platform*" : Proceedings of the TINA 1999 Conference, Oahu, USA '99

[BHW98] Born, Hoffmann, Winkler: "*SDL Enhancements and Integration into the Design-Lifecycle of Telecommunication Services*", Proceedings of the International Conference on Communication Technologies (ICCT'98), Beijing, China '98

[BHW 98a] Born, Hoffmann, Winkler: "*The ITU-ODL to C++ Mapping and its Integration into an SDL based Design-Methodology for Telecommunication Services*", Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC (SAM´98), Berlin, Germany '98

[Bitk 00] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien: "*Informationstechnik und Telekommunikation im Dauerhoch*", Bitkom-Presseinformation, http://www.bitkom.org/presse/pr230200.htm, BitKom '00

[Bitk 00a] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien: "*Wege in die Informationsgesellschaft - Status quo und Perspektiven Deutschlands im internationalen Vergleich*", BitKom '00

[BK 00] Born, Kath: "*From TINA-ODL Towards a Component Oriented Design Method*". Proceedings of the TINA 2000 Conference, Paris, France '00

[BK 00a] Born, Kath: "*Code Generation for Component based Telecommunication Service Development*", Proceedings of the SoftCom 2000 Conference, Split, Croatia '00

[BK 00b] Born, Kath: "*Customizing UML for Component Design*", Proceedings of the 1st OMG Workshop: UML In The .com Enterprise - Modeling CORBA, Components, XML/XMI And Metadata, Palm Springs, USA '00

[BK 01] Born, Kath: "*Distributed Applications: From Models to Components*", EURESCOM Workshop on Middleware in Telecommunications, Kjeller, Norway '01

[BK 01a] Born, Kath: "*The DOT Profile*", http://www.dot-profile.de/

[BK 01b] Born, Kath: "*Distributed Applications: From Models to Components*", Presentation at the Object Management Group TC Meeting, document number telecom/01-03-02, Irvine, USA '01

[BKH 00] Born, Kath, Holz: "*A UML Profile for Integrated Design and Development of Distributed Applications*", Proceedings of the Technology of Object-Oriented Languages and Systems 2000 (TOOLS Pacific 2000) Conference, Sydney, Australia '00

[BKvH00] Born, Kath, v. Halteren: „*Modeling and Runtime Support for Quality of Service in Distributed Component Platforms*", Work in Progress Paper at 11th Annual IFIP/IEEE International Workshop on "Distributed Systems: Operations and Management", Austin, USA '00

[Box99] Box: "*COM - The Component Object Model*", Addison-Wesley '98

[BMBF 00] Bundesministerium für Bildung und Forschung: "*Analyse und Evaluation der Softwareentwicklung in Deutschland*", http://www.dlr.de/IT/IV/Studien/evasoft_abschlussbericht.pdf, BMBF '00

[BS98] Born, Strick: "*Development Tools for Distributed Services*", European Research Consortium for Informatics and Mathematics News No.34, Sophia Antipolis, France '98

[BSH 01] Born, Schieferdecker, Hoffmann: "*The Deployment and Configuration Language*", EURESCOM Workshop on Middleware in Telecommunications, Kjeller, Norway '01

[BSL 00] Born, Schieferdecker, Li: "*Test Framework for Component-Based Systems*", 20th International Conference on Distributed Computing Systems (ICDCS' 2000) with International Workshop on Distributed System Validation and Verification (DSVV'2000), Taipei, Taiwan '00

[BSL 00a] Born, Schieferdecker, Li: "*UML Framework for automated Generation of component based test systems*", 1st International Conference on Software Engineering Applied to Networking & Parallel/ Distributed Computing (SNPD'00), Reims, France '00

[CCMP00] Research Project Description: "*Implementation of CORBA Component Model Session Container*", Berlin, Germany '00

[ComPoTel 00] Research Project Description: "*Component Oriented Design in Telecommunications - Concepts, Notation and Middleware Platform Support*", TINA Fellowship Program, Red Bank, USA '00

[*CoRE* I] Born, Kath: "*CoRE - Komponentenorientierte Entwicklung offener, verteilter Softwaresysteme im Telekommunikationskontext, Band I - Entwicklungsprozesse und Entwicklungstechniken*"

[*CoRE* II] Born: "*CoRE - Komponentenorientierte Entwicklung offener, verteilter Softwaresysteme im Telekommunikationskontext, Band II: Konzeptraum und Notationen*"

[*CoRE* III] Kath: "*CoRE - Komponentenorientierte Entwicklung offener, verteilter Softwaresysteme im Telekommunikationskontext, Band III: Plattformunterstützung und Ableitungsregeln für Softwarekomponenten*"

[DBAG98] "*AEROSPACE*", Magazine of Daimler-Benz Aerospace AG, 1/98, Stuttgart, Germany '98

[DBB+ 01] Dubois, Born, Boehme, Fischer, Holz, Kath, Neubauer, Stoinski: "*Distributed Systems: From Models to Components*", Proceedings of the 10th SDL Forum Conference, Copenhagen, Denmark '01

[DSTCdMOF] Distributed Systems Technology Centre: "*dMOF - An OMG Meta Object Facility Implementation*", http://www.dstc.edu.au/products/CORBA/MOF

[EUP715] EURESCOM Project P715 Deliverable: "*Deliverable 2: Experiments on the EURESCOM Services Platform, Final Report*", Heidelberg, Germany '99

[EUP910] EURESCOM Project P910 Deliverable: "*Deliverable 7: Report on Telecommunication Application Domains*", Heidelberg, Germany '01

[EUP924] EURESCOM Project P924: "*Deployment and configuration support for distributed PNO applications*", http://www.eurescom.de/public/projects/P900-series/p924/P924

[EUP924a] EURESCOM Project P924 Deliverable: "*Deliverable 2: Notation and semantics for deployment and initial configuration*", Heidelberg, Germany '01

[FBH+ 99] Fischbeck, Born, Hoffmann, Winkler, Baudis, Böhme, Fischer: "*SDL enhancements and application for the design of distributed services*": Proceedings of the 9th SDL Forum Conference, Montreal, Canada '99

[FFB 98] Fischer, Fischbeck, Born: "*SDL und ODL im Entwicklungsprozess von Telekommunikationssystemen*": König, Langendörfer (eds.), Formale Beschreibungen für Verteilte Systeme, Shaker Verlag, Aachen, Germany '99

[FFH+ 97] Fischbeck, Fischer, Holz, Kath, v. Löwis, Schröder: "*Improving the Development and Validation of Viewpoint Specifications*", Proceedings of the Conference on Formal Methods for Object Oriented Distributed Systems (FMOODS) '97, Centerburry, UK '97

[FHK+ 98] Fischbeck, Holz, Kath, Vogel: "*Flexible Support of ORB Interoperability*", Proceedings of the Interworking '98 Conference, Ottawa, Canada '98

[FK 98] Frolund, Koistinen: "*QML: A Language for Quality of Service Specification*", White Paper, Hewlett-Packard Laboratories '98

[FK 99] Fischbeck, Kath: "*CORBA Interworking over SS.7*", Proceedings of the Conference on Intelligence in Services in Networks, Barcelona, Spain '99

[FKB 00] Fischer, Kath, Born: "*TINA-ODL and Component Based Design*", Proceedings of the 6th International Conference on Object-Oriented Information Systems 2000, London, UK '00

[FKH+ 98] Fischbeck, Kath, Holz, Geipl, Vogel: "*Operational and Stream Interactions in a B-ISDN based Environment*", Proceedings of the INFORMS 1998 Conference, Boca Raton, USA '98

[FTK+00] Funabashi, Toyouchi, Kanai, Uchihashi, Kobayashi, Hakomori, Yoshida, Strick, Born: "*Development of Open Service Collaborative Platform for Coming ECs by International Joint Efforts*" Procceedings of International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet, Rom, Italy '00

[Gart99] Gartner Executive Report: "*Application Integration: Putting the Pieces of the Puzzle Together*" Gartner Group '99

[Gart99a] Gartner Strategic Analysis Report: "*Middleware Deployment Trends: Survey of Real-World Enterprise Applications*" Gartner Group '99

[GHJ+ 99] Gamma, Helm, Johnson, Vlissides: "*Elements of Reusable Object-Oriented Software*", Addison-Wesley '99

[HKG+ 97] Holz, Kath, Geipl, Lin, Vogel: "*The CAMOUFLAGE Project -Introduction Of TINA Into Telecommunication Legacy Systems*", Proceedings of the TINA 1997 Conference, Santiago de Chile, Chile '97

[HWB 97] Hoffmann, Winkler, Born, Fischer, Fischbeck: *"Towards a Behavioural Description of ODL"* Proceeding of the TINA 1997 Conference, Santiago, Chile '97

[HV 99] Henning, Vinoski: "*Advanced CORBA Programming with C++*", Addison-Wesley '99

[InstallShield] InstallShield Software Corporation: "InstallShield 6.3", http://www.installshield.com/

[ITUTX.292] ITU-T Recommendation X.292: *"OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – The Tree and Tabular Combined Notation (TTCN)"*, ITU '98

[ITUTX.608] ITU-T Recommendation X.608: "*Abstract Syntax Notation One*", ITU-T '99

[ITUTX.902] ITU-T Recommendation X.902 |ISO/IEC 10746-2: *"Open Distributed Processing - Reference Model Part 2"*, ITU-T/ISO '95

[ITUTX.903] ITU-T Recommendation X.903 | ISO/IEC 10746-3: *"Open Distributed Processing - Reference Model Part 3"*, ITU-T/ISO '95

[ITUTX.904] ITU-T Recommendation X.904 | ISO/IEC 10746-4: *"Open Distributed Processing - Reference Model Part 4"*, ITU-T/ISO '95

[ITUTZ.100] ITU-T Recommendation Z.100: "*Specification and Description Language*", ITU-T '00

[ITUTZ.109] ITU-T Recommendation Z.109: "*SDL combined with UML*", ITU-T '00

[ITUTZ.130] ITU-T Recommendation Z.130: "*The ITU-T Object Definition Language*", ITU-T '99

[Kath 99] Kath: "*How to get Behavioural Information on CORBA Systems?*" Proceedings of the Workshop on the Application of Distributed Object Technology, Heidelberg, Germany '99

[KHF+97] Kath, Holz, Fischer, Geipl, Vogel: "*Provision of TINA Object Interaction through B-ISDN in ATM networks*", Proceedings of the GLOBECOM Conference, Phoenix, USA '97

[KKS00] Kath, Kleinschmidt, Stoinski: *"CPE Architecture for TINA Services (CATS II): Final Project Deliverable"*, document number CATS-II HU 006, Tokio, Japan '00

[KS00] Kath, Stoinski: "*Project Proposal: Platform for Authoring and Composition of Interactive Content (PACIFIC)*", Projektvorbereitungsdokument, Berlin, Germany '00

[KST+ 01] Kath, Stoinski, Takita, Tsuchiya: "*Middleware Platform Support for Multimedia Content Provision*", Proceedings of the 3G Technologies and Applications Conference, Heidelberg, Germany '01

[KT 99] Kath, Takita: "*OMG A/V Streams and TINA NRA: An integrative Approach*", Proceedings of the TINA 1999 Conference, Oahu, USA '99

[KvHS+ 00] Kath, v. Halteren, Stoinski, Wegdam, Fisher: "*Middleware Platform Management Based on Portable Interceptors*", Proceedings of 11th Annual IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Austin, USA '00

[MDTS 01] Forschungs- und Entwicklungsprojekt: "*Model Driven Development of Distributed Telecommunication Systems*", Bundesministerium für Bildung und Forschung, Berlin, Germany '01

[MICO] Römer, Puder et. al.: "*MICO for C++*", http://www.mico.org/

[MS .net] Microsoft Corporation: "*The .NET Framework*", Micosoft Developer Network Online, http://msdn.microsoft.com/net/framework/default.asp

[MS C#] Microsoft Corporation: "*C# Introduction and Overview*", Micosoft Developer Network Online, http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp

[MS COM] Microsoft Corporation: "*The Component Object Model*", Microsoft Press '98

[Neu 95] Neumann: "*Objektorientierte Entwicklung von Software-Systemen*", Addison-Wesley '95

[OMGASRFP] Object Management Group: *"Action Semantics for the UML Request For Proposal"* OMG document ad/98-11-01

[OMGAVStreams] Object Management Group: "*Control and Management of Audio/Video Streams*", OMG document formal/00-01-03

[OMGCCM I] BEA Systems et. al.: *"CORBA Components - Volume I"*, OMG document orbos/99-07-01

[OMGCCM II] BEA Systems et. al.: "*CORBA Components - Volume II*", OMG document orbos/99-07-02

[OMGCCM III] BEA Systems et. al.: "*CORBA Components - Volume III*", OMG document orbos/99-07-03

[OMGCCM RFP] Object Management Group: "*CORBA Component Model RFP, Final Version*", OMG document orbos/97-06-12

[OMGCORBA] Object Management Group: „*The Common Object Request Broker: Architecture and Specification, Revision 2.4.2*", OMG document formal/01-02-33

[OMGCORBAES] Object Management Group: "*Event Service Specification, Version 1.1*", OMG document formal/01-03-01

[OMGCORBAIDL] Object Management Group: „CORBA 2.4.2 IDL Syntax and Semantics", OMG document formal/01-02-39

[OMGCORBANaS] Object Management Group: "*CORBA Naming Service*", OMG document formal/01-02-65

[OMGCORBANoS] Object Management Group: "*Notification Service Specification, Version 1.0*", OMG document formal/00-06-20

[OMGCORBAP] Data Access Corporation et. al.: "*UML Profile for CORBA*", OMG document ad/00-02-02

[OMGCWM] Object Management Group: "*Common Warehouse Metamodel (CWM) Specification, Version 1.0*", OMG document ad/01-02-01

[OMGC++Map] Object Management Group: "*C++ Language Mapping Specification*", OMG document formal/99-07-41

[OMGDeplRFP] Object Management Group: "*Deployment and Configuration of Distributed Applications Draft RFP*", OMG document orbos/01-04-12

[OMGMDA] Object Management Group: "*Model Driven Architecture*", OMG document omg/00-11-05

[OMGMICRFP] Object Management Group: "*Multiple Interfaces and Composition RFP*", OMG document orb/96-01-04

[OMGMOF1.3] Object Management Group: "*Meta Object Facility, Version 1.3*", OMG document formal/00-04-03

[OMGMOFP] Object Management Group: "*UML Profile for MOF*", OMG document ad/01-02-29

[OMGPI] BEA Systems et. al.: "*Portable Interceptors*", OMG document orbos/99-12-02

[OMGPSS2.0] Object Management Group: "*Persistent State Service 2.0*", OMG document orbos/99-07-07

[OMGReqUMLP] Object Management Group: "*Requirements for UML Profiles*", OMG document ad/99-12-32

[OMGSPEM 01] IBM, Rational Software, Fujitsu/DMR, SOFTEAM, Unisys, Nihon Unisys Ltd., Alcatel, Q-Labs: "*Software Process Engineering Management: The Software Process Engineering Metamodel (SPEM)*", OMG document ad/2001-03-08

[OMGSPERFP] Object Management Group: "*Software Process Engineering (SPE) Management Request for Proposal*", OMG document ad/99-11-04

[OMGUML1.3] Object Management Group: "*OMG Unified Modeling Language Specification, Version 1.3*", OMG document ad/99-06-08

[OMGXMI1.1] Object Management Group: "*XML Metadata Interchange (XMI) version 1.1*" OMG document formal/00-11-02

[OOC] Object Oriented Concepts Inc.: "*ORBacus™ for C++ and Java*", http://www.ooc.com/ob/

[OOCa] Object Oriented Concepts: "*Customer Success Stories*", http://www.ooc.com/customers/telecom.html

[OOCJTC] Object Oriented Concepts Inc.: "*Java™-like Threads for C++*", http://www.ooc.com/jtc/

[Pom 91] Pomberger: "*Prototyping-Oriented Software Development - Concepts and Tools*", Structured Programming, Vol. 12, Nr. 1 '91

[PS 94] Pagel, Six: "*Software Engineering - Band 1: Die Phasen der Softwareentwicklung*", Addison-Wesley '94

[Rational] Rational Software Corp.: "*Customer Stories*", http://programs.rational.com/success/Success_Result.cfm

[RationalRose] Rational Software Corp.: "*Rational Rose*", http://www.rational.com/products/rose/index.jsp/

[RationalRUP] Rational Software Corp.: "*Rational Unified Process: Best Practices for Software Development Teams*" Rational Software Corp. White Paper, http://www.rational.com/products/whitepapers/100420.jsp

[Ray95] Raymond: "*Reference Model of Open Distributed Processing (RM-ODP): Introduction*" Proceedings of International Conference of Open Distributed Processing, Brisbane, Australia '95

[RFCMIME] Internet RFC 2046: "*Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*", IETF '98

[Roy 70] Royce: "*Managing the Development of Large Scale Software Systems*", Proceedings of WESCON Conference, Los Angeles, USA '70

[SB97] Strick, Born: "*Developing TINA Services with ODL and SDL*" Proceedings of the Workshop on Distributed Object Technologies for Telecoms Networks (DOT '97), Heidelberg, Germany '97

[Sun 01] Sun Microsystems White Paper: „*Network Application Strategies for Telecom: Introduction Component Architectures for Enhanced Services*", http://www.sun.com/software/solutions/third-party/software/whitepapers/Java.Telco.pdf

[Sun EJB] Sun Microsystems: "*Enterprise JavaBeans TM*" http://www.sun.com

[Sun EJBa] Sun Microsystems: "*Industry Opinions On Enterprise JavaBeansTM (EJBTM) vs. COM+/MTS*", http://java.sun.com/products/ejb/ejbvscom.html

[Sun JAVA] Sun Microsystems: "*Java 2 Platform Standard Edition*", http://www.javasoft.com/j2s

[Sun SSL] Sun Microsystems, Netscape Corp.: "*Introduction to SSL*", http://docs.iplanet.com/docs/manuals/security/sslin/index.htm

[Szy99] Szyperski "*Component Software - Beyond Object-Oriented Programming*", Addison-Wesley '99

[TINA] TINA-C: "http://www.tinac.com"

[TINACMC] TINA-C: "*Computational Modeling Concepts*",TINA-C '98

[TINAReq] TINA-C: "*Requirements upon TINA-C Architecture*", TINA-C '95

[TINASA] TINA-C: "*TINA Service Architecture 5.1*", TINA-C '97

[TTK+ 00] Tsuchiya, Takita, Kath, Stoinski: "*Authoring, Composition, and Delivery of Interactive Contents by the use of Multimedia Contents Mill*", Proceedings of the TINA 2000 Conference, Paris, France '00

[TTK+ 00a] Tsuchiya, Takita, Kath, Stoinski: "*'Multimedia Contents Mill' – A Platform for Authoring and Delivery of Interactive Multimedia Contents*", Proceedings of the SoftCom 2000 Conference, Split, Croatia '00

[UnisysUREP] Unisys Corp.: "*Universal Repository*", http://www.unisys.com/marketplace/urep/

[vHalt00] A.T. van Halteren: "*A reflective QoS provisioning service for object middleware*", Workshop on Reflective Middleware (RM 2000), co-located with the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '00), New York, USA '00.

[VSB+ 99] Vassiliou-Gioles, Schieferdecker, Born, Winkler, Li: "*Configuration and Execution Support for Distributed Tests*" Proceedings of the 12th International Workshop on Testing of Communicating System (IWTCS 99), Budapest, Hungary '99

[Web01] Weber: "*Quo Vadis Software Engineering*", Gesellschaft für Informatik, Softwaretechnik-Trends, Band 21 Heft 1, http://pi.informatik.uni-siegen.de/stt/21_1/index.html, GI '00

[WS 96] Weck, Szyperski: „*Do We Need Inheritance?*" Proceedings of the Workshop on Composability Issues in Object-Orientation (at ECOOP'96), Linz, Austria '96

[WWF+ 95] Wasowski, Witaszek, Fischer, Holz, Lau, Kath: "*Co-Simulation of Hybrid SDL and VHDL Specifications*", Proceedings of the 9th ESM Conference, Prague, Chzech Republic '95

[W3CHTTPNG] Janssen et. al.: „*HTTP-ng Architectural Model*", W3C Internet Draft, www.w3c.org

[W3COSD] Marimba Incorporated, Microsoft Corporation: "*The Open Software Description Format (OSD)*", submitted to W3C, http://www.w3.org/TR/NOTE-OSD.html, W3C '97

[W3CSMIL] W3C Recommendation: "*Synchronized Multimedia Integration Language (SMIL 2.0) Specification* ", W3C Proposed Recommendation, http://www.w3.org/TR/2001/PR-smil20-20010605/, W3C '01

[W3CXML] W3C Recommendation: "*Extensible Markup Language (XML) 1.0 (Second Edition)*", http://www.w3.org/TR/2000/REC-xml-20001006, W3C '00

[W3CXMLDT] W3C Recommendation: "*XML Schema Part 2: Datatypes*", http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/, W3C '01

ABKÜRZUNGEN

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CCM	CORBA Component Model
CDR	Common Data Representation
CIDL	Component Implementation Definition Language
CIF	Component Implementation Framework
CO	Computational Object
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CoRE	Components Rapid Engineering
DCOM	Distributed COM
DTD	Document Type Definition
EJB	Enterprise Java Beans
HTTP-NG	HTTP Next Generation
IDL	Interface Definition Language

IIOP	Internet Inter-ORB Protocol
OCL	Object Constraint Language
ODL	Object Definition Language
OMG	Object Management Group
OSD	Open Software Description
MDA	Model Driven Architecture
MOF	Meta Object Facility
MP3	MPEG Audio Layer-3
MPEG	Moving Picture Experts Group
RFP	Request for Proposals
RMI	Remote Methode Invocation
RM-ODP	Reference Model for Open Distributed Processing
RUP	Rational Unified Process
SDL	Specification and Description Language
SPEM	Software Process Engineering Metamodel
TINA	Telecommunications Information Networking Architecture
UDP	User Datagram Protocol
UML	Unified Modeling Language
UUID	Universal Unique Identifier
XMI	XML Metadata Interchange
XML	Extensible Markup Language

INDEX

ANHANG A *Eingeführte Termini*

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Artefakt	*artifact*	Abstraktion konkreter programmiersprachlicher, in Softwarekomponenten in Form von Codemodulen enthaltener Konstrukte (z.B. Klassen im Falle von objektorientierten Programmiersprachen) Artefaktinstanzen realisieren das Verhalten, den Zustand und die Identität von COs	CCM
Assemblage	*assembly*	Beschreibung eines verteilten Softwaresystems durch Angabe der beteiligten CO-Typen und deren initialer Konfiguration	CCM
Attribut (eines Interfacetyps)	*attribute*	Spezielle Variante von Operationen im Sinne einer verkürzten Schreibweise für *get*- und *set*-Operationen für einen bestimmten Datentyp	CORBA
Ausführungszeit	*runtime*	Zeit, in der die in Softwarekomponenten enthaltenen Codemodule durch eine Maschine ausgeführt werden	
Ausnahme	*exception*	Spezielle Art der Operationsterminierung im Falle von Fehlern	RM-ODP

Tab.2 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Bindung	*binding*	Konzept zur Beschreibung des Austausches von Referenzen auf Instanzen der zu *Port*-Definitionen gehöhrenden Interfacetypen unter Beachtung der Art der *Port*-Definition (*Used*- oder *Provided*-*Port*-Definition)	RM-ODP, CCM
Bindungsfall	*binding case*	Zusammenfassung von COs, Bindungen dieser COs und zu diesen Bindungen gehörigen Bindungsregeln	
Bindungsregel	*binding rule*	Beschreibung von geforderten Eigenschaften von Interaktionen, die auf der Grundlage von Bindungen entstehenden, formuliert als Regeln über Kontrakttypen	RM-ODP
Component-Support-Plattform	*component support platform*	Plattform, die *Deployment*- und Ausführungsunterstützung für Softwarekomponenten unter Benutzung einer *Distributed-Processing*-Umgebung bereitstellt	
Computational-Objekt	*computational object*	Objekt, das durch funktionale Dekomposition eines Softwaresystems während dessen Modellierung entsteht	RM-ODP, TINA
Computational-Objekttyp	*computational object type*	Beschreibung von *Computational*-Objekten, die zu deren Instanziierung benutzt wird	RM-ODP, TINA
Consume-Definition	*consume*	Interaktionselement der Signalinteraktion, beschreibt durch Angabe eines Signaltyps die Möglichkeit des Konsumierens eines Signals dieses Signaltyps im Kontext eines Interfaces	CCM
Continuous-Media-, Signal- und operationale Interaktion	*continous media, signal and operational Interaction*	Interaktion zwischen COs unter Verwendung von operationalen, Signal- oder *Continous-Media*-Interaktionselementen (Operation, Attribut, *Consume*, *Produce*, *Sink*, *Source*)	RM-ODP, TINA (operationale und *Continous-Media*-Interaktion)
Datentyp	*data type*	Element eines Datentypsystems (z.B. CORBA-IDL) und Basis von Informationsmodellen, dient der strukturierten Repräsentation von Information	CORBA
Distributed-Processing-Umgebung	*distributed processing environment*	Technologische Grundlage zur Unterstützung der Interaktion von Objekten eines verteilten Softwaresystems	TINA
Implementierungs-element	*implementation element*	Beschreibung einer Relation zwischen Interaktionselement und Artefakt mit der Bedeutung, daß eine Instanz des Artefakts für die Realisierung des Verhaltens des Interaktionselements verantwortlich ist.	
Implements-Relation	*implements*	Relation zwischen Artefakten und CO-Typen mit der Bedeutung, daß Instanzen der Artefakte das Verhalten der COs dieses CO-Typs realisieren	

Tab.2 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
initiales CO	*initial CO*	Instanz eines CO-Typs, die zu Beginn der Ausführungszeit des Softwaresystems erzeugt wird	CCM (nicht als CO sondern als CORBA *component* bezeichnet)
initiale Konfiguration	*initial configuration*	Beschreibung der initialen COs und deren initialen Bindungen	CCM
initiale Bindung	*initial binding*	Bindung, die zu Beginn der Ausführungszeit eines Softwaresystems erzeugt wird	CCM
Instanziierungs-muster	*instantiation policy*	Beschreibung, nach welchen Regeln zur Ausführungszeit Instanzen des durch ein Artefakt modellierten programmiersprachlichen Konstruktes angelegt werden sollen	
Interaktionselement	*interaction element*	Generalisierung der Konzepte Operation, Attribut, *Sink*, *Source*, *Consume*, *Produce*	
Interface	*interface*	auf der Grundlage eines Interfacetyps im Kontext eines COs entstehende referenzierbare Zusammenfassung von potentiellen Interaktionen eines COs	RM-ODP
Interfacetyp	*interface type*	Aggregation von Interaktionselementen als benannter, identifizierbarer Endpunkt von potentieller Interaktion	RM-ODP
Kontrakttyp	*contract type*	Datentyp, der Elemente zur Definition von Eigenschaften der aufgrund von Bindungen zustandekommenden Interaktionen beinhaltet	QML
Konzeptraum	*concept space*	Die Gesamtheit aller für die objektorientierte Modellierung in einem Anwendungsgebiet verwendbaren Konzepte und deren Beziehungen	
Maschine	*node*	Gerät, das zur Ausführung der in Softwarekomponenten enthaltenen Codemodule geeignet ist (i.allg. Computer)	RM-ODP
Medienmenge	*media set*	Aggregation von Medien	
Medientyp	*media type*	Vorschrift für die Codierung, die Übertragung und Decodierung der Mediendaten eines Mediums	
Medium	*medium*	Abstraktion einer multimedialen Information	
Metamodell	*metamodel*	Modell, daß das Instrumentarium zur Definition von Modellen beschreibt	MOF
Modellklasse	*viewpoint*	Definition einer Einschränkung auf bestimmte Aspekte von Entwurfsmodellen durch Angabe von hierfür relevanten Konzepten und Beziehungen eines Konzeptraumes	RM-ODP

Tab.2 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Multiple-Port-Definition	*multiple port*	*Port*-Definition, auf deren Grundlage eine Vielzahl von Interfacereferenzen zur Ausführungszeit eines COs hinterlegbar oder beschaffbar ist	CCM (dort nur für benutzte Interfacetypen)
Namensraum	*namespace*	Konzept zur Strukturierung von Namen der Elemente eines Modells	RM-ODP
Objekt	*object*	Modell einer Entität von Interesse in der betrachteten Anwendungsdomäne	RM-ODP
Objektumgebung	*environment of an object*	Teil des Softwaresystems, der nicht Bestandteil des Objektes ist	RM-ODP
Operation	*operation*	Interaktionselement der operationalen Interaktion, beschrieben durch Parameter und mögliche Arten der Terminierung	RM-ODP, CORBA
Parameter	*parameter*	Identifizierbarer Bestandteil einer Operation, definiert die Richtung des Informationsflusses bei der Interaktion und einen der Information zugrundeliegenden Datentyp	CORBA
Port-Definition	*port*	Beschreibung der Möglichkeit der Hinterlegung bzw. Beschaffung von Referenzen auf Interfaces eines entsprechenden COs zur Ausführungszeit	CCM
Prädikat	*predicate*	Abstraktion von zu wahr oder falsch evaluierbaren Ausdrücken	RM-ODP
Produce-Definition	*produce*	Interaktionselement der Signalinteraktion, beschreibt durch Angabe eines Signaltyps die Möglichkeit des Versendens eines entsprechenden Signals im Kontext eines Interfaces	CCM
Provided-Port-Definition	*provides*	auf einer *supports*-Relation basierende *Port*-Definition zur Beschreibung der Möglichkeit der Beschaffung von Referenzen auf Interfaces eines COs zur Ausführungszeit	CCM
Realize-Relation	*realize*	Zuordnung von CO-Typen zu Softwarekomponenten	UML
Requires-Relation	*requires*	Relation zwischen Interfacetyp und CO-Typ mit der Bedeutung, daß COs dieses CO-Typs Interfaces dieses Interfacetyps von ihrer Umgebung erwarten	TINA
Sicht	*viewpoint*	s. Modellklasse	RM-ODP
Signal	*signal*	Atomare Nachricht, die asynchron und entkoppelt zwischen COs ausgetauscht wird, entsteht auf der Basis eines Signaltyps	RM-ODP

Tab.2 Verwendete Termini in der Übersicht

Konzept	englische Bezeichnung	Erklärung	Bezug zu anderen Ansätzen und Standards (falls vorhanden)
Signalparameter	*signal parameter*	Identifizierbare Angabe eines Datentyps im Kontext eines Signaltyps	RM-ODP
Signaltyp	*signaltype*	Beschreibung von Signalen, die zu deren Instanziierung im Kontext von Signalinteraktion verwendet wird, Aggregation von Signalparametern	RM-ODP
Single-Port-Definition	*single port*	*Port*-Definition, auf deren Grundlage eine einzelne Interfacereferenz zur Ausführungszeit eines COs hinterlegbar oder beschaffbar ist	CCM
Sink-Definition	*sink*	Interaktionselement der *Continous-Media*-Interaktion, beschreibt durch Angabe einer Medienmenge die Möglichkeit des Empfangs der Medien auf der Basis eines geeigneten Medientyps im Kontext eines Interfaces	TINA
Software-komponente (modelliert)	*software component*	Abstraktion einer Softwarekomponente im Modell (Abstraktion von den Instruktionssequenzen)	[Szy99]
Software-komponente (real)	*software component*	physikalisch repräsentierte Entität, bestehend aus in Codemodulen zusammengefaßten Instruktionssequenzen, die Ausführung einer Softwarekomponente führt zur Inkarnation von Objekten	[Szy99]
Softwarepaket	*software package*	Zusammenfassung von Softwarekomponenten eines realen Softwaresystems	CCM, OSD
Softwaresystem (real)	*software system*	System bestehend aus Softwarekomponenten	[Szy99]
Source-Definition	*source*	Interaktionselement der *Continous-Media*-Interaktion, beschreibt durch Angabe einer Medienmenge die Möglichkeit des Sendens der Medien auf der Basis eines geeigneten Medientyps im Kontext eines Interfaces	TINA
supports-Relation	*supports*	Relation zwischen Interfacetyp und CO-Typ mit der Bedeutung, daß COs dieses CO-Typs Interfaces dieses Interfacetyps ihrer Umgebung bereitstellt	TINA
Terminierung	*termination*	Ende einer gerufenen Operation	RM-ODP
Used-Port-Definition	*uses*	auf einer *requires*-Relation basierende *Port*-Definition	CCM
Zustandsattribut	*state attribute*	spezieller Datentyp zur Repräsentation von Zustandsinformationen von CO-Typen	RM-ODP, CCM

Tab.2 Verwendete Termini in der Übersicht

ANHANG B *Fallbeispiel*

In diesem Anhang werden die Entwicklungstechniken von *CORE* zur Entwicklung eines verteilten Softwaresystems eingesetzt. Damit wird die Anwendung der Konzepte von $CORE_{CEPT}$ zur Modellbildung, die Darstellung des Entwurfsmodells mit den Notationen von $CORE_{TATIONS}$ sowie die Anwendung von $CORE_{MAP}$ zur automatischen Ableitung von Softwarekomponenten aus Modellen demonstriert. Die für diese Entwicklungsaufgabe verwendete Komponentenarchitektur ist $CORE_{WARE}$. Als Entwicklungsprozeß wird hier aufgrund der eingeschränkten Komplexität des Beispiels das klassische Phasenmodell ohne Iterationen verwendet.

B.1 Analyse und Definition

Das zu entwickelnde Softwaresystem soll das klassische Beispiel „*Dining Philosophers*" implementieren. Dieses Anwendungsbeispiel wurde von Edger W. Dijkstra 1965 erstmalig formuliert. Es ist ein Modell und eine universale Methode zum Testen und Vergleichen von Theorien über das Alloziieren von Ressourcen. Das Problem besteht aus einer endlichen Menge von Prozessen die eine ebenfalls endliche Menge von Ressourcen teilen. Die Ressourcen können immer nur von einem Prozeß gleichzeitig verwendet werden - damit können sowohl *Lifelock*- als auch *Deadlock*- Situationen entstehen.

Eine Konkretisierung des Problems kann wie folgt beschrieben werden: Eine Menge von Philosophen versammelt sich um einen Tisch, auf dem eine Menge von Gabeln bereitsteht. Philosophen führen die Aktionen Essen, Schlafen und Denken aus. Zum Essen benötigen sie zwei Gabeln - eine für die rechte Hand und eine weitere für die linke Hand. Die Gabeln werden den Philosophen aus der Menge der verfügbaren Gabeln zugewiesen, d.h. ein Philosoph muß zum essen immer die ihm zugeordneten Gabeln verwenden. Gabeln können nur von jeweils einem Philosophen zu einem Zeitpunkt benutzt werden.

Zur Visualisierung des Verhaltens von Philosophen gibt es einen Observierer (*Observer*). Dieser soll von den Philosophen über die Veränderung ihrer aktuell ausgeführten Tätigkeit informiert werden. Diese Situation ist in Abb.43 verdeutlicht.

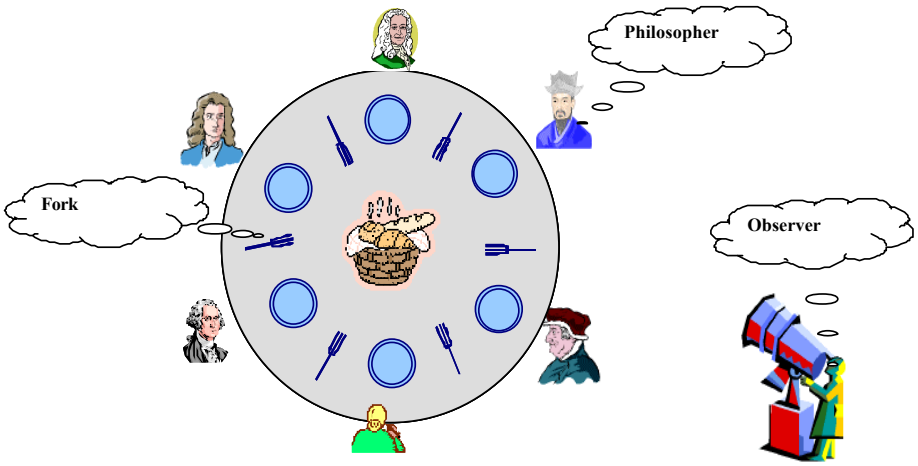


Abb. 43 Das Beispiel „Dining Philosphers“

Das beschriebene Beispiel ist so zu implementieren, daß Philosophen, Observierer und Gabeln auf unterschiedlichen Maschinen ausgeführt werden und trotzdem miteinander interagieren können. Es werden keine zusätzlichen nicht-funktionalen Anforderungen an das Softwaresystem gestellt.

B.2 Entwurf

STRUKTURSICHT. In der Struktursicht sind die Grundelmente für die Interaktionen zwischen den COs eines verteilten Softwaresystems zu spezifizieren.

Für das hier betrachtete Beispiel ist konkret eine Signal-basierte Interaktion zwischen Philosophen und dem Observierer zu modellieren. Hierzu wird ein Signaltyp ***PhilosopherState*** definiert, der als Parameter den Datentyp ***PState*** unter dem Namen ***carry_pstate*** besitzt. Elemente dieses Datentyps sind der Name des Philo-

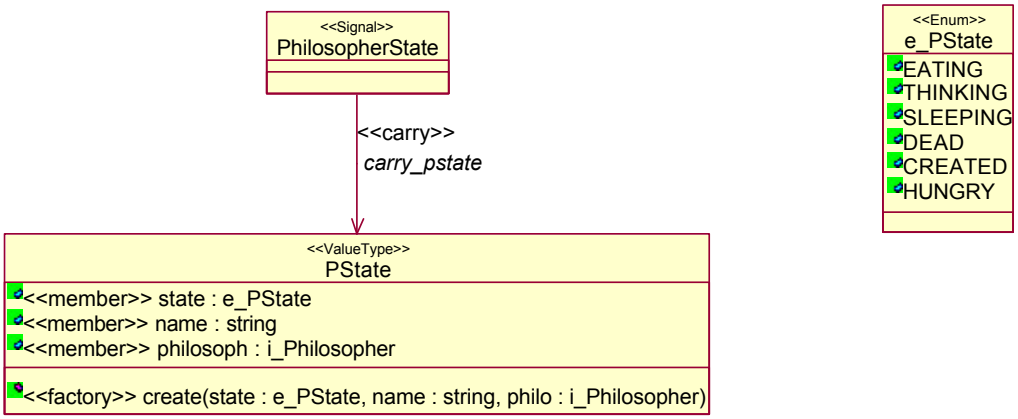


Abb. 44 Signal- und Signalparameter

sophen, der Zustand des Philosophen sowie ein weiteres Element, das als Typ einen Interfacetyp besitzt, dessen Bedeutung später beschrieben wird. Der Zustand eines Philosophen wird durch den Datentyp ***e_Pstate*** beschrieben. Alle diese Modellelemente sind unter Verwendung von *CoRE*$_{TATIONS}$ in Abb.44 notiert.

Auf der Basis der definierten Elemente können nun die Interfacetypen zur Interaktion zwischen COs eingeführt werden (vgl. Abb. 45):

- Der Interfacetyp ***i_Observer*** beinhaltet das *Consume*-Interaktionselement ***pstate*** zum Empfang von ***PhilosopherState***,
- der Interfacetyp ***i_Philosopher*** beinhaltet die Operation ***set_name*** mit dem Parameter ***name*** des Datentyps ***string***,
- der Interfacetyp ***i_Fork*** beinhaltet die Operationen ***obtain_fork*** und ***release_fork***; jeweils mit dem Parameter ***requester*** vom Typ ***o_Philosopher*** zur Identifikation des Philosophen, der eine Gabel benutzen oder freigeben möchte. Der Operation ***obtain_fork*** ist die Ausnahme ***ForNotAvailable*** zugeordnet, die ausgelöst wird, falls eine Gabel zum Zeitpunkt des Operationsrufes nicht verfügbar ist, d.h. von einem anderen Philosophen benutzt wird. Der Operation ***release_fork*** ist die Ausnahme ***NotTheEater*** zugeordnet, die ausgelöst wird, falls der Rufer dieser Operation nicht der aktuelle Besitzer der Gabel ist.

Für dieses Beispiel werden Philosophen, Gabeln und Observierer durch die CO-Typen ***o_Philosopher***, ***o_Observer*** und ***o_Fork*** modelliert.

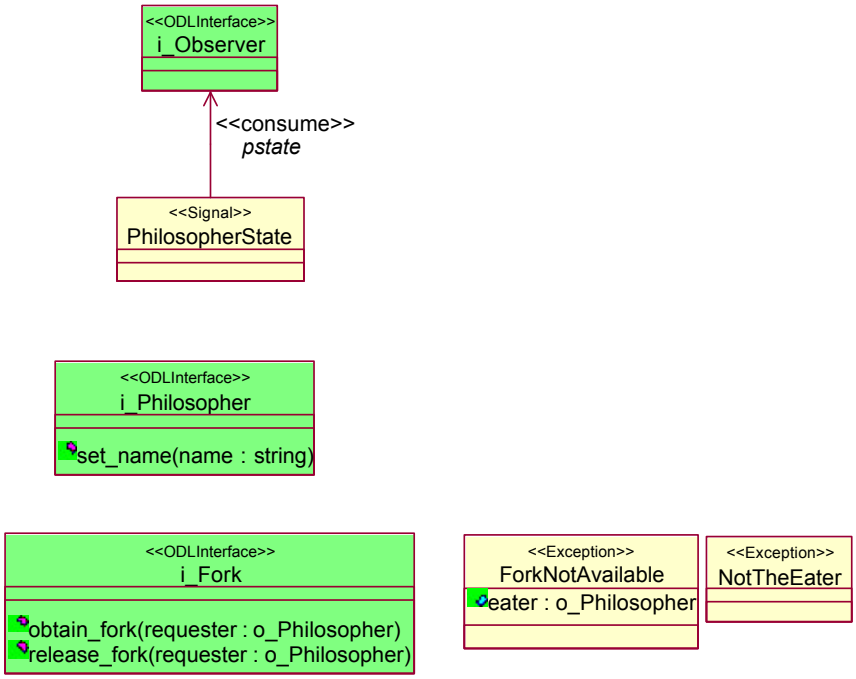


Abb. 45 Interfacetypen

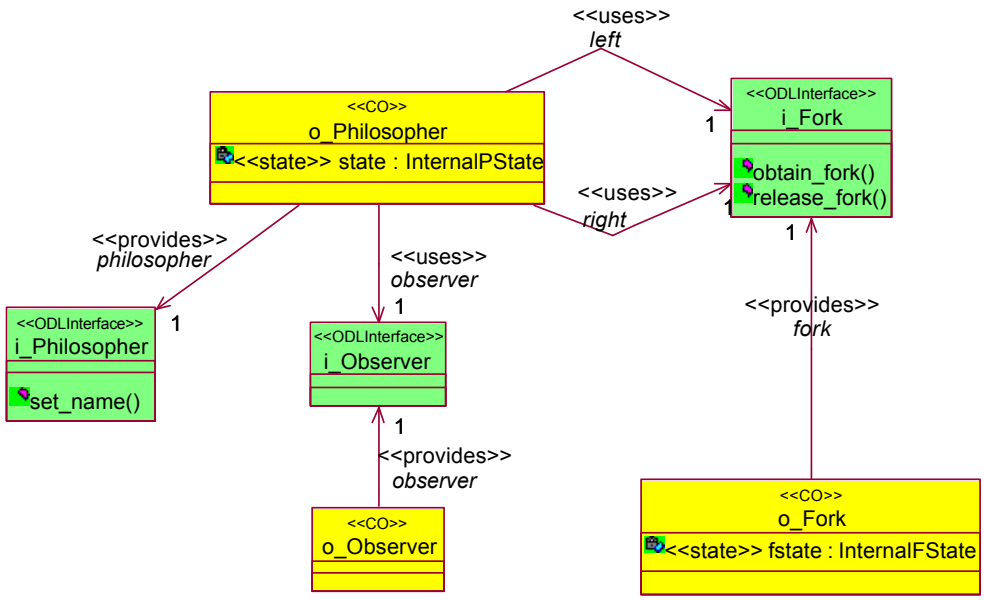


Abb. 46 CO-Typen

KONFIGURATIONSSICHT. Für die definierten CO-Typen werden folgende *Single-Port*-Definitionen vorgenommen (vgl. Abb. 46):

- ***o_Fork*** enthält eine *Provided-Port*-Definition mit dem Namen ***fork***, an der der Interfacetyp ***i_Fork*** angeboten wird,
- ***o_Observer*** enthält eine *Provided-Port*-Definition mit dem Namen ***observer***, an der der Interfacetyp ***i_Observer*** angeboten wird,
- ***o_Philosopher*** enthält eine *Provided-Port*-Definition mit dem Namen ***philosopher***, an der der Interfacetyp ***i_Philosopher*** angeboten wird,
- ***o_Philosopher*** enthält zwei *Used-Port*-Definitionen mit den Namen ***left*** und ***right***, an denen der Interfacetyp ***i_Fork*** genutzt wird,
- ***o_Philosopher*** enthält eine *Used-Port*-Definition mit dem Namen ***observer***, an der der Interfacetyp ***i_Observer*** genutzt wird.

IMPLEMENTIERUNGSSICHT. Die CO-Typen werden durch Artefakte wie folgt realisiert (vgl. Abb. 47):

- ***o_Fork*** wird durch das Artefakt ***o_Fork_Impl*** mit den Implementierungselementen ***obtain_fork_impl*** und ***release_fork_impl*** für die Interaktionselemente ***obtain_fork*** und ***release_fork*** realisiert,
- ***o_Observer*** wird durch das Artefakt ***o_Observer_Impl*** mit dem Implementierungselement ***pstate_in*** für das Interaktionselement ***pstate*** von ***i_Observer*** realisiert,
- ***o_Philosopher*** wird durch das Artefakt ***o_Philosopher_Impl*** mit den Implementierungselementen ***set_name_impl*** für das Interaktionselement ***set_name*** von ***i_Philosopher*** und ***pstate_out*** für das Interaktionselement ***pstate*** von ***i_Observer*** realisiert.

Der CO-Typ ***o_Philosopher*** erhält ein Zustandsattribut ***state*** vom Typ ***InternalPState***, der CO-Typ ***o_Fork*** ein Zustandsattribut ***fstate*** vom Typ ***InternalFState***.

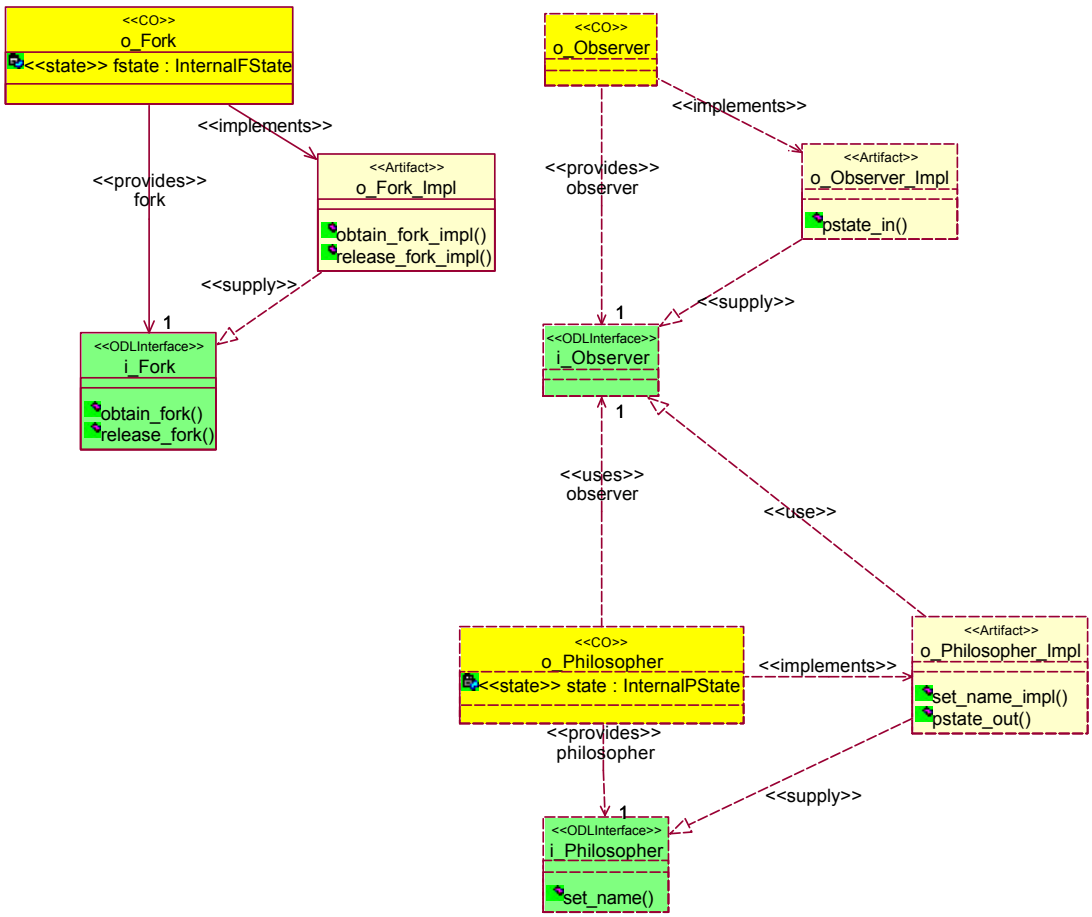


Abb. 47 Artefakte

DEPLOYMENT-SICHT. Die CO-Typen werden Softwarekomponenten wie folgt realisiert (vgl. Abb. 48)

- ***o_Philosopher*** und ***o_Fork*** werden durch die Softwarekomponente ***Philosopher*** bereitgestellt,
- ***o_Observer*** wird durch die Softwarekomponente ***Observer*** realisiert.:

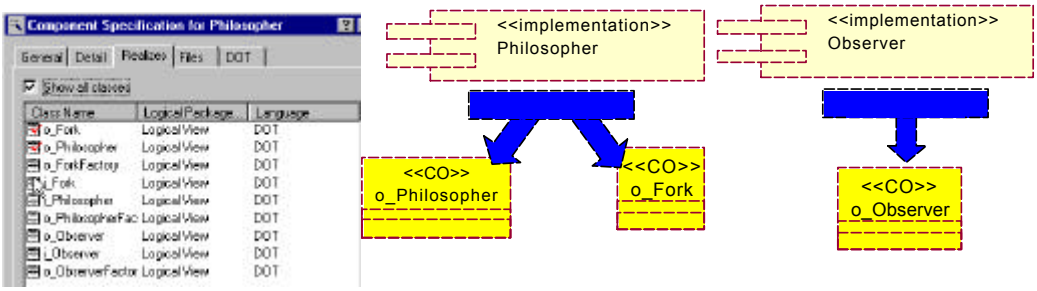


Abb. 48 Softwarekomponenten

INTERAKTIONSSICHT. Da keine spezifischen nicht-funktionalen Anforderungen an das zu entwickelnde Softwaresystem in der Analyse- und Definitionsphase gestellt wurden, sind keine diesbezüglichen Entwurfsdefinitionen notwendig.

B.3 Implementierung

Durch *CORE*$_{MAP}$ werden CORBA-IDL-Interfacedefinitionen und C++-Codemodule erzeugt. Die Codemodule werden vom Entwickler komplettiert. Folgende Implementierungsaktivitäten sind dabei auszuführen:

- Implementierung der Artefaktklassen für ***o_Observer_Imp***, ***o_Fork_Impl*** und ***o_Philosopher_Impl*** sowie
- die Vervollständigung der main-Methode der entsprechenden CO-Klassen.

Der durch den Entwickler definierte C++-Code ist im folgenden dargestellt.

```
void o_Fork_Impl::obtain_fork_impl
( ::Philosophers::o_Philosopher_ptr requester, const char * name )
 {
  JTCSynchronized sync ( * dynamic_cast < CO_o_Fork * > (co())->state()->fstate() );
  if ( dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->state == USED )
    throw ForkNotAvailable ();
  dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->state = USED;
  dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->eater
    = ::Philosophers::o_Philosopher::_duplicate ( requester );
  unsigned long my_number
    = dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->number;
  GuiControl::instance()->use_fork(my_number, name );
 }

 void o_Fork_Impl::release_fork_impl
 ( ::Philosophers::o_Philosopher_ptr requester )
 {
  JTCSynchronized sync ( *dynamic_cast < CO_o_Fork * > (co())->state()->fstate() );
  if ( dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->state == USED )
  {
    dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->state = UNUSED;
    dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->eater
      = ::Philosophers::o_Philosopher::_nil();
    unsigned long my_number
      = dynamic_cast < CO_o_Fork * > (co())->state()->fstate()->number;
    GuiControl::instance()->unuse_fork(my_number);
    Sleep ( 250 );
  }

 void o_Philosopher_Impl::set_name_impl ( const char * name )
 {
  JTCSynchronized sync ( * dynamic_cast<CO_o_Philosopher *> (co())->state()->state() );
  dynamic_cast < CO_o_Philosopher * > (co())->state()->state()->name
    = CORBA::string_dup ( name );
 }

 void CO_o_Philosopher::main ()
 {
  GuiControl::instance()->inc_philosopher();
  srand( (unsigned)time( NULL ) );
  ::Container::Container::instance()->message ( "CO_o_Philosopher::main()" );
  {
    JTCSynchronized sync ( *state()->state() );
    state()->state()->name = CORBA::string_dup ( "PHILOSOPH" );
  }

  {
    JTCSynchronized sync ( *this );
```

```
  while ( CORBA::is_nil ( get_left() ) || CORBA::is_nil ( get_right () ) )
  {
   this->wait();
  }
}

try
{
  ComponentModel::Parameters params;
  this->resolve_port_supply ( "::Philosophers::o_Observer", "observer", params );
  CORBA::Object_var provided_
    = this->resolve_port ( "::Philosophers::o_Observer", "observer" );
  this->co()->connect ( "observer", provided_, params );
  }
catch ( ... )
{
  MESSAGE ( "cannot connect to observer" );
}

send_signal ( this, CREATED );
while ( 1 )
{
  send_signal ( this, HUNGRY );

  bool have_left_fork = false;
  bool have_right_fork = false;
  while ( ! have_left_fork || ! have_right_fork )
  {
    if (!have_left_fork)
    {
      try
      {
        get_left()->obtain_fork ( co(), CORBA::string_dup ( state()->state()->name) );
        have_left_fork = true;
      }
      catch ( ForkNotAvailable& )
      {
        if ( have_right_fork )
        {
          MESSAGE ( "release right fork" );
          get_right()->release_fork ( co() );
          have_right_fork = false;
        }
          Sleep ( rand() % 100 );
      }
    }

    if (!have_right_fork)
    {
      try
      {
        get_right()->obtain_fork ( co(), CORBA::string_dup ( state()->state()->name) );
        have_right_fork = true;
      }
      catch ( ForkNotAvailable& )
      {
        if ( have_left_fork )
        {
          get_left()->release_fork ( co() );
          have_left_fork = false;
        }
          Sleep ( rand() % 100 );
      }
    }
  }
    if ( observer_ )
```

```
      send_signal ( this, EATING );
    Sleep ( rand() % 2000 );
    try
    {
      get_left() -> release_fork ( co() );
      get_right() -> release_fork ( co() );
    }
    catch ( NotTheEater& )
    {
      MESSAGE ( "someone has stolen the fork" );
    }
    send_signal ( this, THINKING );
      Sleep ( rand() % 2000 );
    if ( observer_ )
      send_signal ( this, SLEEPING );
      Sleep ( rand() % 2000 );
  }
  send_signal ( this, DEAD );
}
```

Für diese Codemodule ist eine Integration mit einer graphischen Benutzeroberfläche vorzunehmen.

B.4 Integration und Test

Nachdem die Softwarekomponenten ***Observer*** und ***Philosopher*** auf den Ausführungsmaschinen bereitgestellt wurden, können mit Hilfe des in $CORE_{WARE}$ enthaltenen generischen Konfigurationswerkzeuges verschiedene Konfigurationen des Softwaresystems aufgebaut und getestet werden. Dabei wird die durch $CORE_{MAP}$ unterstützte Konfigurierbarkeit der Softwarekomponenten ausgenutzt (vgl. Abb. 49).

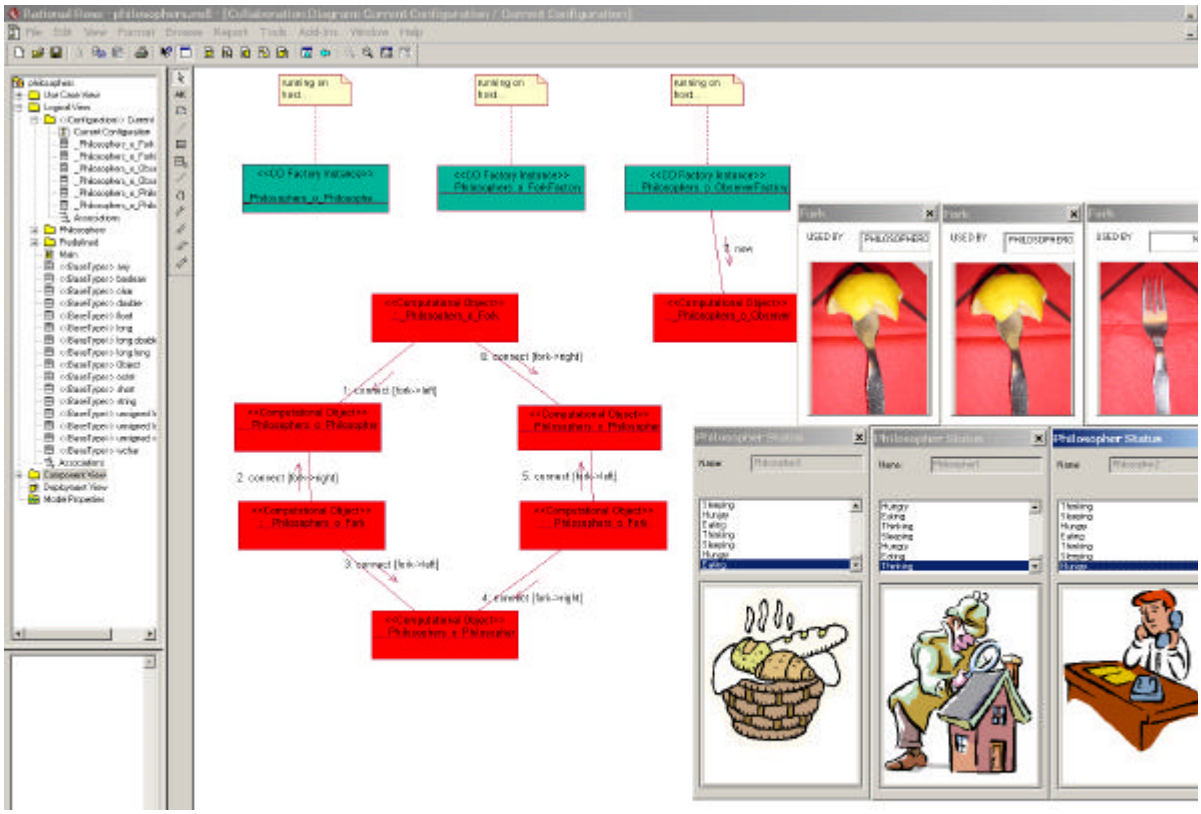


Abb. 49 Integration und Test

ANHANG C

Ableitungsregeln in der Übersicht

Regel	Erklärung
Regel 1	Ableitungsregel für das Konzept Namensraum nach CORBA-IDL
Regel 2	Ableitungsregel für Spezialisierungen von Datentyp nach CORBA-IDL
Regel 3	Ableitungsregel für das Konzept Ausnahme nach CORBA-IDL
Regel 4	Ableitungsregel des Konzeptes Signalparameter auf den CORBA-IDL-Datentyp **valuetype**
Regel 5	Ableitungsregel für das Konzept Signaltyp auf den CORBA-IDL-Datentyp **valuetype**
Regel 6	Erweiterung der Ableitungsregel für das Konzept Signaltyp um eine **factory**-Operation im erzeugten CORBA-IDL-Datentyp **valuetype**
Regel 7	Ableitungsregel von Generalisierungsrelationen zwischen Instanzen des Konzeptes Signaltyp
Regel 8	Ableitungsregel zur Produktion der Spezifika von **factory**-Operationen für Signaltypen, die in Generalisierungsrelation stehen
Regel 9	Ableitungsregel für die Konzepte Interfacetyp und operationales Interaktionselement sowie für Definitionen, die in Instanzen des Konzeptes Interfacetyp enthalten sein können, nach CORBA-IDL
Regel 10	Ableitungsregel zur Erzeugung von CORBA-IDL-Moduldefinitionen für Instanzen des Konzeptes Interfacetyp mit nicht-operationalen Interaktionselementen

Tab.3 Ableitungsregeln von CoRE$_{MAP}$ in der Übersicht

Regel	Erklärung
Regel 11	Ableitungsregel zur Erzeugung von CORBA-IDL-Interfacedefinitionen für *Produce*-Definitionen an Instanzen des Konzeptes Interfacetyp für einen Anbieter dieses Interfacetyps
Regel 12	Ableitungsregel zur Erzeugung einer CORBA-IDL-Interfacedefinition, die die entsprechend *Regel 11* produzierten CORBA-IDL-Interfacedefinitionen in einer Komposition zusammenfaßt
Regel 13	Ableitungsregel zur Erzeugung von CORBA-IDL-Interfacedefinitionen für *Produce*-Interaktionselemente in Instanzen des Konzeptes Interfacetyp für einen Nutzer dieses Interfacetyps
Regel 14	Ableitungsregel zur Erzeugung einer CORBA-IDL-Interfacedefinition, die alle entsprechend *Regel 13* produzierten CORBA-IDL-Interfacedefinitionen mittels Generalisierung zusammenfaßt
Regel 15	Ableitungsregel zur Erzeugung von CORBA-IDL-Definitionen für *Consume*-Definitionen an einer Instanz des Konzeptes Interfacetyp
Regel 16	Ableitungsregel zur Erzeugung von CORBA-IDL-Interfacedefinitionen für *Source*-Interaktionselemente in Instanzen des Konzeptes Interfacetyp für einen Anbieter dieses Interfacetyps
Regel 17	Ableitungsregel zur Erzeugung einer CORBA-IDL-Interfacedefinition, die die entsprechend *Regel 16* produzierten CORBA-IDL-Interfacedefinitionen in einer Komposition zusammenfaßt
Regel 18	Ableitungsregel zur Erzeugung von CORBA-IDL-Interfacedefinitionen für *Source*-Definitionen an Instanzen des Konzeptes Interfacetyp für einen Nutzer dieses Interfacetyps
Regel 19	Ableitungsregel zur Erzeugung einer CORBA-IDL-Interfacedefinition, die die durch *Regel 18* produzierten CORBA-IDL-Interfacedefinition in einer Komposition zusammenfaßt
Regel 20	Ableitungsregel zur Erzeugung von CORBA-IDL-Definitionen für *Sink*-Definitionen an einer Instanz des Konzeptes Interfacetyp
Regel 21	Ableitungsregel zur Erzeugung von CORBA-IDL-Definitionen für in einer Generalisierungsbeziehung stehende Instanzen des Konzeptes Interfacetyp
Regel 22	Ableitungsregel zur Erzeugung einer CORBA-IDL-Interfacedefinition für eine Instanz des Konzeptes CO-Typ und dessen Operationen und Attribute
Regel 23	Ableitungsregel zur Erzeugung von CORBA-IDL-Definitionen für in einer Generalisierungsbeziehung stehende Instanzen des Konzeptes CO-Typ
Regel 24	Ableitungsregel zur Erzeugung von CORBA-IDL-Interfacedefinitionen für CO-Fabriken
Regel 25	Erweiterung von *Regel 24* bezüglich der Spezialisierung der CORBA-IDL-Interfacedefinition **ComponentModel::CoFactoryBase** durch die für CO-Fabriken erzeugten CORBA-IDL-Interfacedefinitionen
Regel 26	Ableitungsregel zur Erzeugung von CORBA-IDL-Definitionen für *Provided-Port*-Definitionen
Regel 27 und *Regel 28*	Erweiterung von *Regel 26* bezüglich der Erzeugung von CORBA-IDL-Definitionen für *Provided-Port*-Definitionen für Interfacetypen mit nicht-operationalen Interaktionselementen
Regel 29 und *Regel 30*	Ableitungsregel zur Erzeugung von CORBA-IDL-Definitionen für *Used-Port*-Definitionen

Tab.3 Ableitungsregeln von CoRE$_{MAP}$ in der Übersicht

Regel	Erklärung
Regel 31 und *Regel 32*	Erweiterung von *Regel 29* bezüglich der Erzeugung von CORBA-IDL-Definitionen für *Used-Port*-Definitionen für Interfacetypen mit nicht-operationalen Interaktionselementen
Regel 33	Ableitungsregel zur Erzeugung einer CORBA-IDL-**valuetype**-Definition für Interfacetypen, die im Kontext einer *Provided-Port*-Definition mit der Auszeichnung *multiple* auftreten
Regel 34	Ableitungsregel zur Erzeugung einer CORBA-IDL-Interfacedefinition für Interfacetypen, die im Kontext einer *Provided-Port*-Definition mit der Auszeichnung *multiple* auftreten, Nutzung der durch *Regel 34* erzeugten CORBA-IDL-**valuetype**-Definition
Regel 35	Erweiterung der durch *Regel 22* produzierten CORBA-IDL-Interfacedefinition für Instanzen des Konzeptes CO-Typ für *Provided-Port*-Definition mit der Auszeichnung *multiple*
Regel 36	Ableitungsregel für *Used-Port*-Definitionen mit der Auszeichnung *multiple*
Regel 37	Ableitungsregel zur Erzeugung von C++-Namensräumen für erzeugte CORBA-IDL-Module
Regel 38	Ableitungsregel zur Erzeugung von C++-*Servant*-Klassen für entsprechend *Regel 9* erzeugte CORBA-IDL-Interfacedefinitionen für den operationalen Anteil von Interfacetypen im Entwurfsmodell
Regel 39	Ableitungsregel zur Erzeugung von C++-*Servant*-Klassen für erzeugte CORBA-IDL-Interfacedefinitionen, die nicht-operationale Interaktionselemente von Interfacetypen des Entwurfsmodells repräsentieren
Regel 40	Ableitungsregel zur Erzeugung von C++-*Servant*-Klassen für CORBA-IDL-Interfacedefinitionen, die CO-Typen des Entwurfsmodells repräsentieren
Regel 41	Ableitungsregel zur Erzeugung von Interface-*Composition*-Klassen für Interfacetypen im Entwurfsmodell
Regel 42	Ableitungsregel zur Erzeugung von lokalen Klassen im Kontext von Interface-*Composition*-Klassen, die den Nutzer bzw. den Anbieter eines Interfacetyps im Entwurfsmodell widerspiegeln
Regel 43	Ableitungsregel zur Erzeugung von C++-Methoden von Interface-*Composition*-Klassen für Interfacetypen, die im Kontext von *Provided-Port*-Definitionen mit der Auszeichnung *multiple* auftreten
Regel 44	Ableitungsregel zur Erzeugung von CO-Typ-*Composition*-Klassen für CO-Typen im Entwurfsmodell
Regel 45	Ableitungsregel zur Erzeugung von *Member*-Definitionen von CO-Typ-*Composition*-Klassen für *Provided-Port*-Definitionen
Regel 46	Ableitungsregel zur Erzeugung von *Member*-Definitionen von CO-Typ-*Composition*-Klassen für *Used-Port*-Definitionen
Regel 47	Ableitungsregel zur Erzeugung von *Member*-Definitionen von CO-Typ-*Composition*-Klassen für *Provided-Port*-Definitionen mit der Auszeichnung multiple
Regel 48 und *Regel 49*	Ableitungsregel zur Implementierung der *Servant*-Klassenmethoden für Interfacenavigation (**provide_<Port-Name>**, **provide_<Port-Name>__Supply**, **connect_<Port-Name>**)
Regel 50	Ableitungsregel zur Implementierung der *Servant*-Klassenmethoden für generische Interfacenavigation (**provide**, **provide_supply** und **connect**)

Tab.3 Ableitungsregeln von CoRE$_{MAP}$ in der Übersicht

Regel	Erklärung
Regel 51	Ableitungsregel zur Implementierung der *Servant*-Klassenmethoden für Interaktionselemente von Interfacetypen (Delegierung an lokale Klassen der Interface-*Composition*-Klassen)
Regel 52	Ableitungsregel zur Erzeugung von C++-Definitionen für Spezifika von *Produce*-Interaktionselementen (Implementierung der entsprechend *Regel 11* erzeugten CORBA-IDL-Interfacedefinitionen)
Regel 53	Ableitungsregel zur Erzeugung von C++-Klassen, die Instanzen des Konzeptes Artefakt im Entwurfsmodell repräsentieren (Artefaktklassen)
Regel 54	Ableitungsregel zur Erzeugung von Methoden der Artefaktklassen, die Implementierungselemente von Artefakten im Entwurfsmodell repräsentieren
Regel 55	Ableitungsregel zur Erzeugung von C++-Klassen für Artefaktfabriken mit den Methoden **`acquire_artifact`** und **`drop_artifact`**
Regel 56	Ableitungsregel zur Erzeugung der Implementierung der Klassenmethoden **`acquire_artifact`** und **`drop_artifact`** (Repräsentation des Konzeptes Instanziierungsmuster)
Regel 57	Erweiterung der Ableitungsregel zur Erzeugung der Interface-*Composition*-Klassen um die abstrakte Klassenmethode **`resolve_artifact`** (Korrelation zwischen Interaktions- und Artefaktmanagement)
Regel 58	Ableitungsregel zur Erzeugung der Implementierung der Methoden der lokalen Klassen der Interface-*Composition*-Klassen (Abbildung der Korrelation von Interaktions- und Implementierungselementen)
Regel 59	Ableitungsregel für die Behandlung von Ausnahmen bei der Korrelation von Interaktions- und Implementierungselementen
Regel 60	Ableitungsregel zur Erzeugung von C++-Klassen als Repräsentationen von CO-Typen im Entwurfsmodell, Spezialisierung der Interface- und CO-Typ-*Composition*-Klassen
Regel 61	Erweiterung der C++-Klasse, die einen CO-Typ des Entwurfsmodells repräsentiert, um die Klassenmethode **`main`**, die nebenläufig ausgeführt wird
Regel 62	Ableitungsregel zur Erzeugung der Implementierung der Methode **`resolve_artifact`** im Kontext der einen CO-Typ repräsentierenden C++-Klasse (diese Methode wird in den Interface-*Composition*-Klassen abstrakt deklariert, die Basisklassen dieser C++-Klasse sind)
Regel 63	Ableitungsregel zur Erzeugung der Implementierung der Interfacenavigation für *Provided-Port*-Definitionen im Kontext von CO-Typ-*Composition*-Klassen
Regel 64	Ableitungsregel zur Erzeugung von C++-Definitionen für das Management von Instanzen von Signal-*Producer*-Klassen für das Produzieren von Signalen
Regel 65	Ableitungsregel zur Erzeugung von C++-Definitionen zur Konfiguration von Signalkommunikationskanälen
Regel 66	Erweiterung der Definition der Artefaktklassen um Notifizierungsmethoden für die erfolgreiche Konfiguration von Signalkommunikationskanälen
Regel 67	Ableitungsregel zur Erzeugung der Implementierung der Interfacenavigation für *Provided-Port*-Definitionen für den nicht-operationalen Anteil der zugehörigen Interfacetypen

Tab.3 Ableitungsregeln von CoRE$_{MAP}$ in der Übersicht

Regel	Erklärung
Regel 68	Ableitungsregel zur Erzeugung der Implementierung der Interfacenavigation für *Used-Port*-Definitionen für den nicht-operationalen Anteil der zugehörigen Interfacetypen
Regel 69	Ableitungsregel zur Erzeugung von C++-Klassendefinitionen für Instanzen des Konzeptes Zustandsattribut
Regel 70	Erweiterung der durch *Regel 69* produzierten C++-Klassen um *Member*-Definitionen für die entsprechend *Regel 64* erzeugten *Management*-Klasseninstanzen für Signal-Produzenten
Regel 71	Erweiterung der CO-Typ-*Composition*-Klassendefinition um *Member*-Definitionen für Zustandsattribute
Regel 72	Ableitungsregel zur Erzeugung der Implementierung von CO-Fabriken
Regel 73	Ableitungsregel für Instanzen der Konzepte Softwarekomponente und *Realize*-Relation
Regel 74	Ableitungsregel für Instanzen des Konzeptes Prädikat
Regel 75	Ableitungsregel zur Erzeugung der Implementierung der Auswertung von Prädikaten zur Ausführungszeit

Tab.3 Ableitungsregeln von CoRE$_{MAP}$ in der Übersicht